

UC Irvine

ICS Technical Reports

Title

Logic, parallelism and semantic networks : the binary predicate execution model

Permalink

<https://escholarship.org/uc/item/5zr3q3f8>

Author

Lee, Craig Alexander

Publication Date

1988

Peer reviewed

2
699
C3
10.88-30

UNIVERSITY OF CALIFORNIA

Irvine

**Logic, Parallelism and Semantic Networks:
the Binary Predicate Execution Model**

Technical Report #88-30

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Information and Computer Science

by

Craig Alexander Lee

Committee in charge:

Professor Lubomir Bic, *Chair*

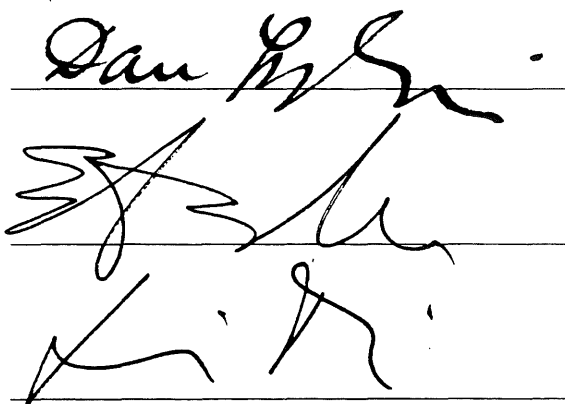
Professor Tatsuya Suda

Professor Daniel Gajski

**Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)**

1988

The dissertation of Craig Alexander Lee is approved,
and is acceptable in quality and form for
publication on microfilm:

Three handwritten signatures are written on three horizontal lines. The top signature is 'Dan W. ...', the middle one is 'J. ...', and the bottom one is 'K. ...'.

Committee Chair

University of California, Irvine

1988

Dedication

This work is dedicated to
Drs. M.B. Meikle and J.A. Vernon
who set my feet on this path.

Contents

List of Tables	<i>vi</i>
List of Figures	<i>vii</i>
Acknowledgements	<i>x</i>
Curriculum Vitae	<i>xi</i>
Abstract	<i>xii</i>
Chapter 1: Why Logic, Parallelism and Semantic Networks?	1
Introduction	1
Organization of the Dissertation	3
Chapter 2: Logic, Parallelism and Semantic Networks	5
Logic and Logic Programming	5
Parallelism	12
Logic and Parallelism	25
Semantic Networks	37
Combining All Three: Logic, Parallelism and Semantic Networks	43
Chapter 3: The Binary Predicate Execution Model.	49
Basic Concepts	49
Clause Invocation	56
Spanning Tree Cost	64
Functional Terms and Variables in the Assertion Graph	67
Non-Logical Extensions	72
Chapter 4: Example Applications	77
Introduction	77
A Geographical Database	77
RMS_KNOMES	82
Chapter 5: Implementation and Architectural Issues	93

Introduction	93
Computation via Message-Passing.	94
Scheduling	109
Allocation.	110
Supporting Architectures	116
Other Issues	120
Chapter 6: Simulations Results	123
Introduction	123
The Simulator	124
OR-Parallelism.	128
Introducing AND-Parallelism.	134
Clause Invocation	136
A Spectrum of Problem Sizes.	142
Independent Routers	148
Conclusions.	150
Chapter 7: Conclusions and Future Research Directions.	152
Contribution	152
Discussion.	152
Related Work	154
Future Work	160
References	162

List of Tables

Table	Page
1. World Geography Database	78
2. Simulation Results under OR-parallelism	130
3. Simulation Results under AND-parallelism	136
4. Simulation Results under Clause Invocation	138
5. Simulation Results for a Spectrum of Sizes	145
6. Simulation Results for a Spectrum with Independent Routers	149

List of Figures

Figure	Page
1. Resolution's Rule of Inference	6
2. A Resolution Example	7
3. Resolvents of Four Clauses	8
4. A Simple Vehicular Semantic Network	38
5. Network Fragment	39
6. Representing Predicates	44
7. An Assertion Graph and Query Template	51
8. A Query Template and One Possible Spanning Tree.	52
9. A Clause Invocation	57
10. Clause Invocation with Binding Order Conflict.	59
11. Invoking Literal as One Vertex.	60
12. General Query Tree.	64
13. Domain Element Fan-in/Fan-out	66
14. Simple Assertion Graph with Functional Terms	68
15. Simple Query with Functional Terms	69
16. Head Unification with Functions from the AG	71
17. An Attribute	73
18. World Geography Database.	79
19. Three Queries in RBL	80
20. Three Clauses in RBL	81
21. The Generic KNOMES Shell	82
22. Form of Semantic Network for RMS_KNOMES.	85
23. Context Clauses Initiated at <i>Cxt</i> Vertices	87
24. Event Clauses Initiated at <i>Scr</i> Vertices.	88

25.	<i>Mdl</i> Clauses for Expected Anomalies	90
26.	<i>Mdl</i> Clauses for Unexpected Anomalies	91
27.	A General Layer of the Activation and Environment Trees	95
28.	An Example of the Activation and Environment Trees	96
29.	Activation Stack Example	99
30.	Activation Stack Path Example	100
31.	Example of Binding Scope	102
32.	Message Grammar	106
33.	Query Message Procedure	108
34.	Success, Failure and Cut Message Procedures	109
35.	A PE with Separate Router	117
36.	Multiple Message-servers	118
37.	Variable Service	119
38.	Simulator Flow Graph	124
39.	The Binary Tree Search Problem	129
40.	Activity Curves for 4x Proportion	130
41.	Activity Curves for 8x Proportion	131
42.	Activity Curves for 16x Proportion	131
43.	Relative Execution Times for each Proportion	132
44.	K-lips for each Proportion	132
45.	Replacing OR-parallelism with AND-parallelism	134
46.	Activity Curves for Different Amounts of AND-parallelism	135
47.	Activity Curves under Simple Invocation	138
48.	Activity Curves under Simple Invocation with OR-parallelism	139
49.	Activity Curves under Nested Invocation	139
50.	Relative Execution Times under Invocation for all Cases	140
51.	K-lips under Invocation for all Cases	140

52.	Activity Curves for $2 \leq h \leq 6$	143
53.	Activity Curves for $7 \leq h \leq 11$	143
54.	Relative Execution Times Approaching Saturation	144
55.	K-lips Approaching Saturation	144
56.	Average Message Queuing for $h = 9, 10$ and 11	146
57.	Maximum Message Queuing for $h = 9, 10$ and 11	146
58.	Average and Maximum Message Queuing for $h = 11$	147
59.	Execution Times with and without Independent Routers.	149

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Lubomir Bic, for his friendship and support of this research. From its initial suggestion to its conclusion, he has always been interested. I also want to thank the other members of my committee, Professors Tatsuya Suda and Daniel Gajski, for their advice and efforts on my behalf.

The "Dataflow" Group, consisting of Jon Gilbert, Mark Nagel, John Roy, Elke Rundensteiner, Wang-Chan Wong and Monika Yin, also deserves recognition. They always provided a good environment for numerous talks, formal and informal, that improved this work.

The entire department and staff of Information and Computer Science at UCI deserve acknowledgement. It was home for many years and the computing environment has continued to grow in impressive ways.

Finally, I want to thank Kathryn Kramer for her patience and companionship throughout this process.

Curriculum Vitae Craig Alexander Lee

Nov. 19, 1952	Born Chicago, Illinois
June 1975	B.A. in Psychology, Reed College, Portland, Oregon
Sept. 1975 – Aug. 1980	Programmer and Research Assistant, Dept. of Medical Psychology and later the Kresge Hearing Research Laboratory, University of Oregon Health Sciences Center, Portland, Oregon
May 1978 – Aug. 1980	Consultant, Portland, Oregon
June 1982	M.S. in Computer Science, Syracuse University, Syracuse, New York
Summer 1983	Visiting Internship, Jet Propulsion Laboratory, Pasadena, California
Sept. 1983 – June 1985	Teaching Assistant, Dept. of Information and Computer Science, University of California, Irvine
Summer 1985	Visiting Lecturer, Dept. of Information and Computer Science, University of California, Irvine
Sept. 1985 – Dec. 1988	Research Assistant, Dept. of Information and Computer Science, University of California, Irvine
Dec. 1988	Ph.D. in Computer Science, University of California, Irvine

Publications

L. Bic and C.A. Lee, "A Data-Driven Model for a Subset of Logic Programming", *ACM Trans. on Programming Languages and Systems*, V9n4 (October 1987) pp. 618-645.

C.A. Lee and L. Bic, "On the Mapping Problem Using Simulated Annealing", *1989 IEEE Int'l. Phoenix Conference on Computers and Communication*, (March 1989) (to appear).

C.A. Lee and L. Bic, "A Parallel Logic Model for Real-Time Knowledge-Base Systems", *First Australian Knowledge Engineering Congress*, (Sept. 1989) (to appear).

Abstract of the Dissertation

Logic, Parallelism and Semantic Networks: the Binary Predicate Execution Model

by

Craig Alexander Lee

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1988

Professor Lubomir Bic, Chair

This thesis develops the Binary Predicate Execution Model; a distributed, massively-parallel system for semantic networks and knowledge bases that is built on a subset of first-order predicate logic. The use of logic gives the model an easily-understood programming paradigm and a well-defined semantics of execution. When expressed in binary predicates, a simple graphical interpretation can be used. All program *facts* are represented in an *assertion graph*. Each vertex is associated with a term appearing in a fact and the edges are labeled with the predicate names. Similar graphs are also associated with each rule body and the query. Finding all possible solutions corresponds to finding all possible matches between the *query graph* and the assertion graph. Invoking a rule corresponds to substituting the graph of its body constrained by the dependencies between its arguments. This can be implemented in a parallel, message-passing fashion where the assertion graph vertices are active processing elements which asynchronously exchange messages identifying different parts of the query that remain to be matched and containing any binding information from previous matching required to accomplish this. The model is data-driven since every message can be immediately processed without the need for any centralized control or centralized memory. By restricting how functional terms can occur, distributed data structures and remote data look-ups for unification are eliminated. Thus, the model's performance on increasingly larger problems *scales-up* given increasingly larger machines in most cases. Architectural support for the model is investigated and simulation results of a relatively simple software implementation are reported. This suggests performance on the order of 10^5 logical inferences per second for 256 processing elements in an n -cube configuration. Further research directions, including that of increasing efficiency, are discussed.

CHAPTER 1

Why Logic, Parallelism and Semantic Networks?

1. Introduction

As the various fields of computer science continue to expand, so do the sizes of the computing problems that need to be solved. Since these problems concern massive amounts of data and since hardware fabrication technology is advancing, parallelism is becoming widely recognized as a viable method for solving these problems in a reasonable amount of time.

Parallelism offers a host of fundamental problems that must be dealt with in any implementation. All computation consists of the *application* of *functions* to *data*. (Whether these functions are *history-sensitive* or not makes the difference between imperative and applicative languages.) Parallelism must decide when and where each application takes place. Since we want machines that are *programmable*, we must decide on a primitive set of functions and allow programs to be data.

Under parallel execution in general, both functions and data can be thought of as entities that move through some physical space with a relative frame of reference. In a von Neumann machine, data (including the program) flows through the von Neumann bottleneck to complete a function application in one, unique processing element (PE). A parallel machine has many PEs. A program can be a chain of instructions or a lattice. A chain of instructions can be sequentially broadcast to multiple PEs handling different data (SIMD). This form of execution is *synchronous* since only a single chain of instructions is being used and the results of each application must be synchronously communicated with the PE that will need it for the next instruction. A lattice can be handled by multiple PEs handling

different instructions and different data (MIMD). Of course, the instruction and data that need each other must be close together and both must be close to some available PE. This can be done by *distributing* the program and data over the PEs and allowing them to execute and communicate *asynchronously*, thus attaining greater parallelism.

The efficient extraction and organization of parallelism from a problem such that this can be done is, of course, the focus of much research. While the hand-coding of parallelism using various language constructs may be appropriate in some cases, the automatic extraction of parallelism that is *inherent* in the expression of the problem promises greater benefit. Such inherent expressions of parallelism can be found in *logic languages*. These languages basically express a problem as disjunctions and conjunctions of literals. Each disjunct represents an independent, alternative solution; doing these in parallel is OR-parallelism. Each conjunct represents a partial solution; doing these in parallel is AND-parallelism. Many efforts for the realization of OR-parallelism and AND-parallelism have been published. While classifications may vary, work has also been reported for *unification*, *search* and *stream*-parallelism. One motivating factor for all this activity is the wide popularity that logic languages are currently enjoying in Artificial Intelligence; a field with no dearth of computationally intense problems amenable to expression in logic.

One of the major tasks of AI is the *representation of knowledge*. This can be as diverse as representing natural language or a set of logical constraints. The *semantic network* is one of the most widely-used knowledge representation tools and has been applied to fields as diverse as natural language and logic. It is basically a graphical entity-relationship hierarchy with the vertices representing concepts and the edges representing relationships between them. By itself, a semantic network is

really only a semantic *memory*; a set of interpretive procedures need to be defined to determine how the network is used and how new information is *inferred* from it.

From the preceding discussion, we see an intersection of these three fields. Semantic networks, including the interpretive procedures, can be expressed in logic which has clear opportunities for parallelism. Any such semantic network based on large amounts of facts offers the potential of massive parallelism. This dissertation is an attempt to realize that potential in the form of the Binary Predicate Execution Model (BPEM). This is a massively-parallel model of computation for a subset of first-order predicate logic that supports the important knowledge representation tool of semantic networks.

2. Organization of the Dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 provides background material and a survey for each of the three areas and their intersections. Logic and logic languages are introduced first, including the definition of unification and resolution. The major issues governing parallelism are then discussed followed by a discussion of the parallelism available in logic. Semantic networks are introduced next. Finally, the intersection of all three defines the area for BPEM.

Chapter 3 introduces the Binary Predicate Execution Model. The transformation of the clausal form of logic into the Assertion Graph and Query Graphs is defined and resolution as a form of graph matching is investigated. Problems concerning functional terms and invoking n -ary clause heads are investigated. Non-logical extensions are also defined.

Chapter 4 gives two larger application examples to illustrate how BPEM can be used. The first is geographical database and the second is a real-time monitoring expert system for the NASA space station remote manipulator.

Chapter 5 investigates the implementation and architectural issues surrounding the realization of BPEM. This develops BPEM's asynchronous, message-passing model of execution, including the binding scope and environment concepts which determine message content. Static allocation is accomplished by simulated annealing. However, a hybrid architecture is proposed to alleviate potential hot-spots.

Chapter 6 presents simulation results that demonstrate BPEM's performance under OR-parallelism, AND-parallelism and several cases of clause invocation. They also demonstrate the property that BPEM's performance on larger problems scales-up given a larger machine in most cases and that simulated annealing works quite well in evenly distributing the problem over the network such that communication costs are low.

Chapter 7 discusses the contribution of the dissertation, related work and future research directions.

CHAPTER 2

Logic, Parallelism and Semantic Networks

1. Logic and Logic Programming

First-order predicate logic is one of the most fundamental fields of mathematics and is involved in all theorem proving. The idea of automating the process of proving theorems is simple and straightforward: given a set of axioms and a rule of inference, it should be possible to *mechanically* derive all theorems based on those axioms, and conversely, given a theorem it should be possible to determine if it is derivable from those axioms. Logic languages are based on modern automatic theorem proving which began with Herbrand in the 1930s. His basic result was that a set S of clauses is unsatisfiable if and only if a finite set S' of ground instances of clauses in S is unsatisfiable. Hence, to test a set of clauses S for unsatisfiability, it is sufficient to mechanically *generate and test* a sequence of finite sets $S'_1, \dots, S'_i, \dots, S'_n$ of ground instances of clauses in S , such that for some finite n , S'_n is unsatisfiable. Unfortunately, successive S'_i generally grow exponentially in size. (A more thorough treatment of all the ideas in this section can be found in Chang and Lee [CL73].)

1.1. Resolution and Unification

An important step in making unsatisfiability testing practical was Robinson's Resolution Principle [ROB65]. To understand resolution, we must introduce the *empty clause*, denoted by \square , which has no conditions and no conclusions. Since the empty clause is unsatisfiable under all interpretations, S is unsatisfiable if it contains or can derive \square . Resolution is used to make this determination. It does so

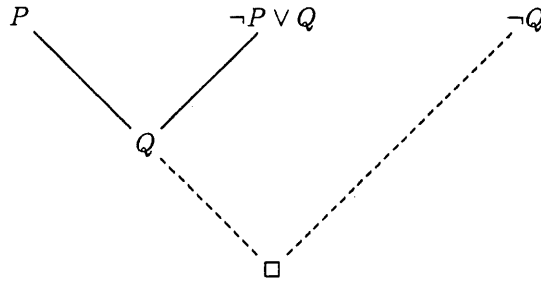


Figure 1

Resolution's Rule of Inference

not by generating and testing sets of ground instances S'_i but by selectively building specific ground instances that derive \square .

Resolution uses a *rule of inference* that is illustrated by the solid lines in Figure 1 and is, for the moment, based in propositional logic. Any two clauses, one with literal P and the other with the complement of this literal, $\neg P$, can be combined to deduce a *resolvent*, after eliminating P and $\neg P$. In this example, the resolvent is Q . The reader may recognize this as *modus ponens* from classical logic. If $\neg Q$ appears in the original set of clauses, then \square can be derived as shown by the dashed lines in Figure 1.

Now consider the literals $P(a)$ and $\neg P(X)$ from predicate logic with the convention that lower-case identifiers are constants and upper-case identifiers are variables. To resolve \square from these two literals, they must be *unified* with the *substitution* $\sigma = \{a/X\}$. To be precise,

Definition: A *substitution* σ is a set of *term/variable* pairs and is written $\{t_1/V_1, \dots, t_n/V_n\}$. The application of a substitution σ to a literal L , written $L\sigma$, is the replacement of every occurrence of V_i in L with t_i for $1 \leq i \leq n$. The empty substitution is denoted by ϵ .

Definition: The *composition* of two substitutions σ and λ is defined by

$$\begin{aligned} \sigma \circ \lambda &= \{t_1/x_1, \dots, t_n/x_n\} \circ \{u_1/y_1, \dots, u_m/y_m\} \\ &= \{t_1\lambda/x_1, \dots, t_n\lambda/x_n, u_1/y_1, \dots, u_m/y_m\} \end{aligned}$$

where any element $t_i\lambda/x_i$ where $t_i\lambda = x_i$ is deleted and any element u_i/y_i where $y_i \in \{x_1, \dots, x_n\}$ is deleted.

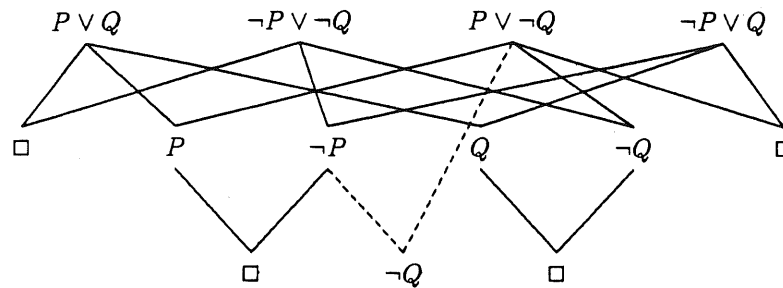


Figure 3

Resolvents of Four Clauses

To illustrate resolution and unification together, we have the little problem below (borrowed from Kowalski [Kow79]):

$$\begin{aligned} & \text{Greek}(\text{socrates}) . \\ & \text{Human}(\text{socrates}) . \\ & \text{Fallible}(Y) \leftarrow \text{Human}(Y) . \\ & \leftarrow \text{Fallible}(X) \wedge \text{Greek}(X) . \end{aligned}$$

The first two lines are facts, the third line is a rule, and the theorem in the last line asks: "Are there any fallible Greeks?" The answer to this is shown in Figure 2. Each pair of edges is a step of resolution where the unifying substitution is shown on the right edge. Not only is the hypothesis shown to be true, the proof is *constructive*; a fallible greek is given by the composite substitution $\{\text{socrates}/X\}$.

1.2. From Theorem Prover to Programming Language

General resolution is fine as a theorem prover but suffers from tremendous inefficiency because there is no specified order of application for the rule of inference. In the search for \square , it is possible to generate irrelevant and redundant proofs, whereby, any set of clauses may have more than one way of deriving \square . Consider Figure 3. The solid lines show four different derivations of \square . The dashed lines indicate one possible beginning of an infinite number of resolutions. By resolving one of the initial clauses with one of the resolvents, it is possible to produce an

infinite sequence of redundant clauses. This subsection shows how resolution can be made the heart of a programming language, such as Prolog, by enabling it to efficiently prove a user theorem or query.

First we note that resolution is *refutation sound* and *complete*. This means that if we wish to prove a specific clause C , we add $\neg C$ to the initial set of clauses and if C is indeed true, we are assured of deriving \square , which denotes falsehood. Second, variants of general resolution that specify order of application have been widely studied (and surveyed) to reduce the number of irrelevant and redundant resolvents [CL73, TC76, Kow79]. The only variant we will present in more detail is that used by conventional Prolog implementations [CM81].

We begin by noting that an arbitrary set of clauses S has a corresponding set of clauses S_H such that S is satisfiable iff S_H is satisfiable. The set S_H consists of *Horn clauses*; clauses with *at most* one uncomplemented literal:

$$A \leftarrow B_1, \dots, B_m .$$

The conclusion of the implication is called the *head*. The conditions of the implication is called the *body*. A clause without a body is called a *fact* or *assertion* because it is logically equivalent to $A \leftarrow \mathbf{T}$ which has the same truth value as A . A clause without a head is called a *goal* or *query* because it is logically equivalent to $\mathbf{F} \leftarrow (B_1, \dots, B_m)$ whose truth value is the negation of the conditions. (This is how a clause is negated to be proven.) A clause with both is also called a *rule*. A set of rules and facts is called a *program* or *knowledge base* against which a query is satisfied.

Conventional Prolog interpreters use a form of *linear input* or *SL-resolution* to satisfy a query, i.e., to find the most general unifier producing a ground instance that deduces \square [EMKo76]. These can be defined as follows:

Definition: *Linear input resolution* is a sequence R_0, \dots, R_i, \dots of resolvents where R_0 is the initial query to be solved and R_i is resolved from R_{i-1} and an input clause, i.e., a fact or rule in the program.

The initial query is a clause body containing only negative literals. Resolution starts by unifying one of these literals with a positive literal which can only be a clause head. The substitutions resulting from this unification are then applied to the query and the associated clause body, if any. If the clause head is part of a rule, then the associated clause body (which a fact does not have) becomes part of the query to be solved. Thus, the number of literals in the query stays the same or increases if the head of a rule is matched and the number of literals decreases if a fact is matched. Resolution continues by unifying query literals with clause heads until, hopefully, \square is derived.

Any resolvent, in general, contains many literals. One literal must be chosen by a *computation rule* [BRUY82] for the next step of resolution but eventually all literals must be satisfied to find a solution. The following specifies the computation rule used in conventional Prolog.

Definition: *SL-resolution* is based on linear input resolution where, for any resolvent $R_i \equiv B_1 \wedge \dots \wedge B_m$, the literal B_1 is selected for possible unification with a program clause $B_1 \leftarrow D_1 \wedge \dots \wedge D_k$ and, as a result,

$$R_{i+1} \equiv D_1 \wedge \dots \wedge D_k \wedge B_2 \wedge \dots \wedge B_m .$$

This clearly suggests the stack implementation of conventional Prolog interpreters.

Any time a literal is chosen from the goal, there may be many facts or heads it could unify with. In this case, the literal is called a *choice point*. One of the candidate facts or clauses must be chosen by a *search rule* [BRUY82] but eventually all must be tried in order to find all possible solutions. In conventional Prolog, these candidates are searched linearly from the first occurrence to the last, i.e., from top to bottom in the program. If the first occurrence unifies and the body is successful, then the original literal is resolved away. If at any later time, the Prolog

interpreter fails to unify a literal (has no chance to derive \square), it will *backtrack* to the most recent choice point, i.e., where there is another occurrence of a literal on which to attempt unification. As a straight-forward extension of this idea, even when a solution has been found, the typical Prolog interpreter can be told to backtrack to the most recent choice point and to continue searching for another solution until all solutions have been found. Hence, the selection of literals and rules forms a tree called the *search space* that the interpreter traverses in the search for solutions where the computations and search rule define the order in which the branches of the search space are investigated which in turn defines the order in which solutions are found.

With SL-resolution, we are always working on the left-most “deepest” literal encountered. This is clearly a *depth-first search* of the search space. It can also be said to be working on one solution at a time. If, however, new clause bodies were added at the end of the current goal list, i.e.,

$$R_{i+1} \equiv B_2 \wedge \dots \wedge B_m \wedge D_1 \wedge \dots \wedge D_k,$$

then a queue implementation would be appropriate. This would be a *breadth-first search*. Many solutions would be underway at once requiring that bindings for distinct solutions be maintained separately.

The depth-first nature of SL-resolution gives logic programs two different “readings”. The first is a *declarative reading*: a logic program is a set of clauses in first-order predicate logic that merely declares the relations between program terms and leaves it up to the interpreter to decide how to satisfy them. The second is a *procedural reading*: a literal is a *procedure call*, whose terms are the actual parameters, and a clause is a *procedure*, whose head defines the formal parameters and whose body is the procedure body. Furthermore, the literals in a procedure body are read (considered for unification) from left to right, top to bottom, just like statements in an imperative language. The procedural reading suggests that

a programmer writes literals as a sequence of activities to be done. It is doubtful that logic languages, like Prolog, would be considered languages at all and would enjoy the acceptance they do today without this feature strongly reminiscent of traditional imperative languages.

To further improve their usefulness, logic languages can diverge from their mathematical basis with a host of *extra-logical* predicates that allows the programmer to control the interpreter and the search space during execution. For example, if the program contains a recursively-defined predicate then the search space contains an infinite branch. When the interpreter reaches this branch, it would continue processing this branch forever and all solutions beyond it would never be found. The *cut* predicate is used to avoid this situation. When encountered, cut instructs the interpreter to remove any choice points, i.e., remove sub-trees in the search space, prior to the cut and including the clause head in which it occurred. Thus, after part of an infinite branch has been processed, a clause containing a cut can succeed, thus telling the interpreter that the recursion is done. As another example, the *assert* and *retract* predicates are used to insert and remove clauses, thereby changing the search space dynamically. Predicates for I/O and 'evaluable' predicates, such as integer addition, also exists. An unfortunate side-effect of this extra control is that the correctness, termination and efficiency of Prolog programs can depend on the order of clauses and the proper use of predicates like cut.

2. Parallelism

The idea of parallelism is that a computational problem can be divided into sub-problems and conquered by many communicating processing elements working simultaneously. The maximum possible benefit to be gained from parallelism is

neatly summed up by Amdahl's Law [AMD67]:

$$S = \frac{1}{s + \frac{p}{n}}$$

where S = relative speed-up
 s = percentage of problem that is inherently serial,
 p = percentage of problem that can be parallel, and
 n = number of PEs.

Clearly, the idea here is that the benefits of parallelism are limited by the parts of a problem that are inherently serial: for $0 < s$ and $p < 1$, $\lim_{n \rightarrow \infty} S = 1/s$. The clear (and fairly obvious) implication is that parallelism is best for problems that contain massive parallelism. An interesting argument was made recently by Gustafson [Gus88] that for many problems, the problem size *and the parallelism* can be scaled-up with the number of PEs such that p can be made arbitrarily close to 1.

Amdahl's Law is, of course, a simplification of the issues governing parallelism. Synthesizing from Treleaven, et al. [TBH82], Vegdahl [VEG84], Gajski and Peir [GP85] and Cvetanović [CVET86], a much more complete set of criteria for judging parallel systems include:

- 1) *Model of Computation*, how the computation is organized.
- 2) *Decomposability* of a problem into sub-problems.
- 3) *Granularity* or size of the sub-problems.
- 4) *Scheduling/Allocation* of the sub-problems to processing elements.
- 5) *Communication/Synchronization* between the processing elements.
- 6) *Network architecture* that the processing elements are built into.
- 7) *Amount of parallelism* in the application problem.

The following subsections look at each of these ideas in more detail.

2.1. Model of Computation

The traditional model of computation is the von Neumann model: a processor uses a program counter to define a sequential *flow of control* between instructions that are fetched from memory. This fetching of instructions one at a time from a memory is called the *von Neumann bottleneck* and is a fundamental limit to the processing speed of this model. Parallelism seeks to avoid this.

Several variations can be done to achieve parallel forms of control flow [TBH82]. The same sequential thread of instructions can be applied to different sets of data, such as vectors, in SIMD computation. This still suffers, however, from the centralized control of one program counter. If we allow multiple program counters exist and follow different threads of execution, we have MIMD computation.

The concept of a program counter, however, can be done away with completely. Instead of one instruction firing another (by “jumping” to it or allowing the program counter to increment), an instruction can execute (fire) when all of its input operands are available. This is called *data-driven computation* and is the basis for *dataflow* computers. Dataflow programs are represented as graphs whose vertices represent operations to be done on *data tokens* that are propagated along the graph edges. When all tokens (operands) are present on the input edges, they are “consumed” and new output tokens are computed and placed on the output edges. Data-driven computation can be called *eager evaluation* because of its fire-when-ready rule.

Instructions can also be fired when there is a specific request for their results. This is called *demand-driven computation* and is the basis for *reduction* computers. Reduction programs are nested expressions and execution can be called *string reduction* or *graph reduction*, depending on whether arguments are copied or shared, respectively. When there is a demand for the program’s results, a demand is created for each part of the nested expression. These demands propagate from the

outermost to the innermost expressions which then return their values. Demand-driven computation can be called *lazy evaluation* because expressions are evaluated only when there is a demand for them.

2.2. Decomposability – Expressing Parallelism

The decomposition of a problem into grains for parallel execution involves not only the partitioning of the problem but also the partitioning of communication to support each sub-problem. While it is common to think about problems “containing” a certain amount of *a priori* parallelism that is decomposable, we must understand that the way a problem is *expressed* affects the possible decomposition and realization of the parallelism that we *think* it contains. Hence, this subsection examines the sources and expressions of parallelism that affect decomposability in the major language classes.

Imperative languages, such as Fortran and Pascal, can include explicit constructs for denoting parallelism. These include *parbegin/parend* [DIK65] (later called *cobegin/coend*) and *fork/join* [DVH66]. Of course, the explicit hand-coding of parallelism with language constructs such as these is limited to the parallelism that the programmer can conceive of. Implicit parallelism in imperative languages, such as parallelism in loops and arrays and also between sections of code that have been serialized due to the nature of these languages, can be extracted by compilers such as Paraphrase and Bulldog. Paraphrase builds a graph expressing the data dependencies in a program and then restructures the graph with specific transformations such that the most parallelism can be utilized when mapped onto an appropriate architecture [PKL80]. Bulldog examines the basic blocks of a program and does microcode compaction over a large trace or execution path through the blocks [FISHER81]. One may conclude from these tactics that imperative languages do not naturally express inherent parallelism.

Functional languages, however, have implicit parallelism that is readily identifiable and explicit parallelism can be an integral part of the language. These languages are based on Church's λ -calculus [CF58] and Curry and Fey's combinatory logic [CF58] and include such classic languages as Lisp [McC60], FP [BAC78] and SASL [TURN79] and also dataflow languages such as Id [AGP78] and Val [McG82]. These languages are based on function application such that the meaning or value of a functional expression only depends on its textual occurrence; this is called *referential transparency*. Functional computation is *ahistoric* in that side-effects and multiple assignment to variables do not exist. The Church-Rosser property [CF58] ensures that the evaluation of a function is the same regardless of the order in which its parts are evaluated. This is to implicitly say that they could be done *in parallel*.

The sources of parallelism in functional languages are no more difficult to find than parsing the program. In contrast, the instruction-at-a-time operational semantics of imperative languages appears to make the expression and extraction of parallelism more difficult. Logic languages also have parallelism that is easily identified with the syntactic structure of their programs. Since this is a central concept in this thesis, a detailed discussion will be postponed until a later section.

2.3. Granularity: Partitioning Work

Once a program has been decomposed into sub-problems for the purposes of parallel execution, the term *granularity* is used to refer to the "size" of the sub-problems. The adjectives *fine* and *coarse* are often used to denote relative the granularities of smaller and larger sub-problems. For example, a system that breaks an arithmetic problem into separate additive and multiplicative operations can be said to have fine granularity. A system whose smallest operation is a complex process can be said to have coarse granularity. The granularity at which a system operates may be related to (1) the programming paradigm, (2) the size of the

application problem, or (3) the size of the system itself. In FP, for example, the finest possible grain naturally suggested by the language would be function application. Multiple applications, however, could be lumped into one grain. Consider that two vectors could be arbitrarily partitioned over a set of PEs for element-wise addition. If the vector length was increased beyond the number of PEs, coarser granularity would be necessary. If the number of PEs was increased, finer granularity would be possible.

While finer granularity may allow more parallel activity, it may not necessarily decrease a program's overall execution time. Decomposing a problem into finer grains usually has a higher cost and then each grain incurs the cost of allocation onto a PE, communicating any initial data to the PE, getting the grain started, and communicating any results which can require synchronization. On the positive side, however, a finer granularity can make *load balancing* easier since it is easier to evenly distribute the total amount of work if the tasks are small [BS81]. Thus, it is a matter of some debate [GP85] whether the allocation, scheduling, communication and synchronization costs of fine-grained dataflow systems [DEN80, AG82, WG82] can be kept from outweighing the benefits of parallelism.

Efforts have been made to adjust granularity to the best level. In the dataflow graphs of Ercegovac, et al., functional primitives are combined together and executed sequentially based on the processing and communication times [ECR84]. Hudak and Goldberg introduce *serial combinators* that, in any given program, are the largest possible combinators that have no concurrent substructure, i.e., can be executed sequentially as one grain, thereby keeping the overhead down while not losing any parallelism [HG85]. Even with these adjustments to granularity, however, it is possible for any finite machine to become swamped with parallelism.

2.4. Scheduling and Allocation

Regardless of how a program is decomposed into grains of parallelism, these grains eventually have to be assigned to PEs for execution. This has been called *scheduling* or *allocation*. While both of these terms have been used to mean the matching of grains to computing resources, we make the following distinction between them. Given a pool of not necessarily independent tasks, scheduling (or a scheduling policy) decides in which order they will execute. Allocation, on the other hand, decides which PEs will execute which grains.

Scheduling was first studied in the 60's and 70's with the advent of time-sharing systems. Many scheduling policies were investigated including first-come-first-serve, round-robin, shortest-job-next, etc. These policies, however, concern pools of unrelated jobs on a single processor. Our concern is related jobs (grains) of the same problem that have definite communication and synchronization constraints on a multi-processor. These constraints can be expressed in a *task precedence graph* or a *task interaction graph* [SE87]. The first is a directed, possibly cyclic graph that defines the partial order in which operations must occur. Dataflow graphs fall into this category. The second is an undirected graph that defines possible process interaction but contains no information about if or when interaction will occur.

Scheduling and allocation in a multi-processor system also addresses the problem of *load-balancing*; the even distribution of work among all PEs. In a multi-processor system, however, this is also constrained by the communication costs between PEs. Tasks that communicate more frequently should be closer to one another. How this problem is approached depends on whether *static* or *dynamic* allocation is used. Static allocation is done at compile-time based on the task graph. While static allocation does not incur any run-time costs, it has the drawback that it cannot adapt to changing conditions over the course of a

computation. Idle PEs cannot be utilized and hot-spots (overloaded PEs) cannot be alleviated. Dynamic allocation addresses this problem but pays the price of run-time costs. It is also usually difficult to implement, especially in a distributed environment, since changing the allocation can mean moving a process with active sub-processes between PEs.

Machine architecture also affects how scheduling and allocation must be done. On a shared-memory or dataflow machine, scheduling and allocation are generally easy: any ready task can be assigned to any idle PE because the communication distances are all the same. Any kind of global scheduler for these architectures, however, is a potential bottleneck when the number of ready tasks becomes large. For a loosely-coupled system, a global scheduler is not practical since the communication costs would certainly be prohibitive. It is these same communication costs that allocation must deal with.

Since optimal scheduling and allocation is NP-Complete [GJ79], most scheduling and allocation techniques are heuristics based on the task graph. Many scheduling techniques use the *critical path* heuristic [PB87, GKS87]. A critical path is found through a graph (by some criteria) and priority is given to the tasks that lay on it. A similar thing is done by trace-scheduling compilers [FISHER81]. These identify a trace or path through the task graph and then reorganize the computation based on those traces.

Specific graph properties can also be exploited. Lee and Aggarwal [LA87], for example, assume prior information about the pattern of communication in the problem graph. Problem edges are then partitioned into *phases* which do not overlap in time. Two edges from different phases can be mapped onto the same network edge since they do not demand bandwidth simultaneously. Hong, Payne and Ferguson [HPF86], however, use information about the structure of the graph. They assume that every graph vertex has either (1) one incoming edge and many

outgoing edges, or (2) many incoming edges and one outgoing edge. Tree-shaped partitions of the graph can then be mapped onto the faces of an n -cube.

Even less information is available, however, when the problem is described by a task interaction graph. Given that the task interaction graph is a very regular *finite element graph*, allocation can be done by 1-D and 2-D strip partitioning that reflects the network topology [SE87]. A similar method assumes that the work load at each vertex is known and recursively bisects the graph such that each resulting mesh represents the same amount of work even though they may not be the same size [BB87]. More importantly, the communication demands across the mesh boundaries are not necessarily the same either. The graph can also be partitioned by using an iterative improvement technique. Bokhari [Bok81] assumes that the problem graph is smaller than the network and represents them as adjacency matrices. The rows and columns of the matrices are then randomly permuted to find a good match of the edges. Simulated annealing [KGV83] can also be used.

Dynamic allocation has been done in certain situations. The Zero Assignment Parallel Processor [BS81] grows processes in a tree structure and allows idle processors to take tasks from neighbors. The Rediflow system [KLT84] uses a "pressure" model to spread tasks over a network. In both cases, child processes are never allowed to be far away from the parent processor to keep communication costs from becoming prohibitive.

2.5. Communication and Synchronization

Communication and synchronization are two closely-related operations in any parallel system. How they are accomplished depends on whether the system is tightly-coupled with a shared-memory or loosely-coupled via message-passing [AS83].

In a shared-memory system, communication is accomplished simply by sharing variables. Once variables are shared, however, they become a resource whose access must be controlled since random reads and writes can corrupt the shared data [CHP71]. This is done by solving the *critical section problem* using various synchronization primitives. One of the most primitive pairs of synchronization primitives are Dijkstra's P and V [DIJK65], also called *signal* and *wait*, which operate on *semaphore* variables. A more structured approach is the use of *conditional critical regions* [HOARE71]. *Monitors* are a further improvement in that related critical regions can be handled in a unified manner [DIJK71, HOARE74]. All of the constraints on the execution of such related regions can be collected into one place by using *path expressions* [CH74].

In a loosely-coupled system, communication is accomplished by sending messages between PEs. This is implemented by *send/receive* primitives with a named destination or source [HOARE78]. This type of communication implies synchronization since a message cannot be received before it is sent. This is affected by whether the send/receive primitives are *blocking* or *non-blocking*. Synchronous message-passing occurs when a send or receive blocks until its matching operation is executed. This usually the case when there is no buffer of data between the PEs. Buffering allows the send to be asynchronous; at least until the buffer is full. Receives are usually blocking since a PE may have nothing to do otherwise but a non-blocking receive can be useful for testing the presence of a message. Blocking send/receives can accomplish the same task as non-blocking ones by using Dijkstra's *guarded commands* [DIJK75].

Aside from the mechanics of communication and synchronization, message-passing systems must minimize the amount of data that must be communicated between parts of the system. That is to say, we have to determine which data must travel between PEs and memories and which can be kept in one place. An

important observation is that *static* data – data that does not change during a computation – should be initially allocated once, while *dynamic* data must, by its nature, be communicated after it is generated during a computation.

Once data has been generated, there must be a decision how it is to be communicated among those PEs that need it. Depending on the system configuration, the data could be *copied* or *shared*. It is not difficult to make n copies of data that are sent to n different PEs. This, however, incurs the cost of copying the data, the cost of communication, and increases the system-wide memory requirements. While sharing data avoids these costs, the shared access must be managed such that data does not become corrupted by competing writers.

2.6. Network Architectures

Multiprocessor architectures can be broadly categorized into tightly-coupled, shared-memory systems and loosely-coupled, message-passing systems. In the first, processors and memories are connected by an *interconnection network*. In the second, processing elements are connected in a certain *network topology*. In both cases, the network properties can determine the overall machine performance. We now review the important parameters of these architectures that are unique to each and those that are common to both such as *cost* and *scalability*, i.e., the ease with which a given network can be made larger. While these architectures are of most interest to us, we will also briefly review broadcast networks.

Interconnection networks connect elements from different types of resources regardless of their function. If there is only one type, then the interconnection network is said to be one-sided; only elements of the same type are connected. An example of this is the public telephone system. We will clearly have at least two types: processors and memories. The parameters that characterize their behavior include:

- 1) *Delay* between the input and output, and

- 2) *Blocking* when two inputs compete for a path through the network.

For example, the *crossbar switch* is used to build *crossbar networks* that provide *non-blocking* communication with constant delay. Since n^2 switches are required for n inputs and n outputs, crossbars are generally considered prohibitively expensive [BH83]. The *shuffle-exchange* concept uses cascaded stages of 2x2 crossbar switches to build *Omega networks*. These are blocking and have a delay of $O(\log_2 n)$ but require only $\frac{n}{2} \log_2 n$ exchange or switch units. Another general class of networks is called the *banyan networks*. They are defined as partially ordered graphs that provide a unique path between every pair of vertices partitioned into two disjoint subsets [CM82, PAT81].

If PEs are limited to “neighbor” connections instead of being able to communicate with any other PE through an interconnection network, we have what are called here *message-passing* networks. These have a different set of parameters that characterize their behavior that includes

- 1) *degree*, the number of edges incident on a vertex,
- 2) *diameter*, the maximum distance in edges between any two vertices,
- 3) *vertex and edge symmetry*, whether the network has no boundaries and “looks” the same from any vertex or edge.

An ideal network would have a low diameter, such that no two vertices are far apart, and a low degree, such that there is not an excessive number of communication channels. Unfortunately, the diameter and degree are, in general, inversely proportional. For example, a *square grid* or *torus* of n vertices with *end-round connections* has a constant degree of 4 regardless of n and a diameter of \sqrt{n} . An *n-cube* of 2^n vertices, on the other hand, has a degree and a diameter of n . A *binary tree* network of n vertices with a height of h , however, has degrees 2, 3, or 1 depending on whether the vertex is the root, internal or a leaf, respectively, and has a diameter of $2h = 2 \log_2(n + 1) - 2$. The symmetry of a graph affects how a problem should be mapped on to it. Both the grid and the n-cube are symmetric

and many mappings on to these graphs would be isomorphic. The tree, however, is not symmetric. Any mapping must minimize the communication required across the root since this is a communication bottleneck. Scalability is concerned with building larger networks. The grid and tree can be scaled-up using the same vertex building block but scaling-up an n -cube requires increasing the number of connections at each vertex. Many different variations of these basic network topologies have appeared in the literature that attempt to combine the best traits of all of them [AL82, GS81, PV81, HG87, AK86, AHK87].

We can generalize message-passing networks to *broadcast* networks by allowing PEs to transmit data to more than one other PE at a time. This is easily accomplished by the use of bus architectures. Broadcast networks potentially reduce the communication time between PEs by reducing the "path" length. A disadvantage is that each PE must constantly listen to the bus to detect data intended for it. Furthermore, each bus is a resource for which there will be contention. The ability of one PE to talk to many is useful in systems designed around a *central controller* [TLMS84] or that have a master-slave relationship between the PEs [TK84]. The central controller idea is the basis of SIMD machines which include vector machines. While useful, this is not an inherent property of broadcast networks. Warren, et al., use a simple bus with PEs of the same class to achieve a practical, realizable system [WADK84]. Borgwardt considers the use of the CM* bus network for his parallel Prolog implementation [BORG84].

2.7. Domains with Massive Parallelism

Clearly we wish to apply parallelism to problems in domains with ample opportunity for it. Here we look at some domains suitable for parallelism; two of which are suitable for BPEM.

Numerical computation over an n -dimensional space of data contains obvious, regularly-structured parallelism. This is the domain of vector machines and array

processors that can efficiently process regularly-structured data. These problems include engineering structural analysis, weather predication and statistics. One interesting example is that of three-dimensional Fourier transforms needed for fusion research. Since BPEM is based on logic and symbolic reasoning, it is not suitable for this type of computation.

Databases are also prime candidates for parallelism since they typically involve massive amounts of data. They can also include rules for reasoning about the data. Most importantly, logic provides an excellent formalism for dealing with databases [GAMI78, WARR81]. Every query that is put to it defines a search space over the facts and the rules of deduction in which the query could have many solutions. Hence, definite databases with deductive laws in Horn clause form are a natural application for BPEM.

In the quest to model more and more complex models of intelligence, the field of Artificial Intelligence poses larger and larger computing problems. While the nature of these problems is varied, Fahlman, et al. [FHS83], give a partial list of basic recurrent computational problems in AI that would benefit from parallelism: Set Intersection, Transitive Closure, Contexts/Partitions, and Pattern Recognition. Solutions to these problems depend on the underlying *knowledge representation*. One important knowledge representation in AI is the *semantic network*. Semantic networks can be defined such that they are a syntactic variant of the clausal form of logic [DK79] and can consist of large amounts of data and rules. Thus, semantic networks can be very similar to the definite deductive databases discussed previously and are also a natural application for BPEM.

3. Logic and Parallelism

Having introduced logic and parallelism separately, we now wish to discuss their intersection. We will first introduce the sources of parallelism in logic and

then survey some of the approaches that have been used to harness it in terms of language constructs and models of execution.

3.1. Sources of Parallelism

3.1.1. AND/OR-Parallelism

As already mentioned, logic languages also have parallelism that is easily identified with their syntactic structure. This is the case for AND-parallelism and OR-parallelism which are best explained together. As discussed previously, the solution to any Horn clause logic program can be described as a tree called the search space. The root is a *conjunction* of literals that corresponds to an initial query and has a child for each literal. Each child, however, corresponds to the *disjunction* of the facts and rules that it (the literal) may unify with. In turn, each rule body is a conjunction of literals and so on. This is, of course, the AND/OR tree description of Horn clause programs that has been widely used [Kow79, CK81, FNM82, CON83, LP84, NAKA84, KM85, LG85]. AND-vertices and OR-vertices occupy alternate tree levels much like a normal form for propositional formulae.

As noted, a literal succeeds if any of its matching facts or rules succeed. A fact always succeeds (when unified) but a rule succeeds only if the entire body does. Each matching rule, however, can be processed in parallel because each is an independent sub-problem. This is called *OR-parallelism*: the parallel search for *alternative* solutions. The original literal is solvable if any of the rules are solvable. As we shall see in the literature, this is the easiest kind of parallelism to realize.

Also as noted, a rule succeeds only if all literals in the body succeed. This is complicated by the possibility of *shared variables* appearing in more than one literal. This introduces a *dependency* or *consistency requirement* that the satisfying substitutions for two different literals must not specify contradictory substitutions for a variable that appears in both. Any parallelism that exists in the derivation

of rule body solutions is called *AND-parallelism*: the parallel search for *partial* solutions. There are at least two methods of realizing AND-parallelism.

The first method solves all literals in parallel as if they were truly independent [TLMS84]. Then a relational equi-join operation is done on the results to extract the consistent substitutions. A similar method involves keeping track of goal-lists that are "AND-branching" where each goal-list has a possibly different set of substitutions [NAKA84]. The AND-branching goal-lists can be joined to form one new list if the substitutions for shared variables do not conflict. This technique involves wasted work (substitutions that are not used) and extra work (determining which substitutions are usable).

The second method defines a *producer-consumer* relation between all literals that share a variable. One literal produces or generates a substitution for a variable and passes this information to all other literals that need it [CG81, CON83, BORG84, IM84]. A similar technique is to assign a priority to each literal whereby the literal with the highest priority gets to specify the substitution for a shared variable prior to any other literal with a lower priority [NAKAG84]. In both cases, a *partial-order* is defined that restricts parallelism but does not waste work making inconsistent substitutions. This could be called *restricted* AND-parallelism but one could also argue that this method does not restrict parallelism but rather identifies the parallelism that exists in a problem. Of course, work must be done to define this partial-order.

3.1.2. Unification Parallelism

This is a blanket term for sources of parallelism that some authors treat separately. Chu and Itano [CI84] identify at least four parallel operations in this classification:

- 1) Parallel searching for rules in the knowledge base.
- 2) Parallel matching of more than one term in a predicate.

- 3) Parallel instantiation of more than one variable.
- 4) Parallel uninstantiation of all the related variables.

Since *search-parallelism*, *(un)instantiation-parallelism* and another special case of unification-parallelism called *stream-parallelism* are relatively implementation dependent, they will be discussed later. For now, we will concentrate on a narrower definition of unification-parallelism, i.e., finding the most general unifier Θ between two literals in parallel.

Unification means finding a substitution that makes all terms in two literals identical. Since a literal can have many terms and unification is at the heart of a logic language, there is clear motivation to find some parallelism. As previously shown, variables can be shared between terms such that unification has a consistency requirement much like AND-parallelism. Ito and Masuda deal with this in their unification data flow graphs by having the “unify” operators feed results into “consistency check” operators [IM84]. The technique used by Yamaguchi, et al. [YTK85], however, partitions the set of terms such that no variables are shared between partitions. Each partition is then conventionally unified. These two approaches to unification parallelism are quite similar to those for AND-parallelism.

Unfortunately, these efforts at unification-parallelism may have limited success. Dwork, et al. [DKM84], have shown that unification is log-space complete for P, the class of languages recognizable in deterministic polynomial time. In terms of parallel complexity theory [Cook83], this is analogous to showing a problem to be NP-complete, i.e., no parallel algorithm may significantly improve on the best sequential algorithm. On a positive note, Dwork, et al., show that a useful special case of unification can benefit from parallelism. This special case is term matching where a term x matches term y if y is a direct substitution instance of x .

3.1.3. Search Parallelism

It can be argued that searching a knowledge base for facts and rule heads that potentially match a query literal is part of unification. Most authors, however, treat search-parallelism separately [CK81, BORG84, TK84, TLMS84]. The parallelism available here, of course, depends on the implementation: if the knowledge base is distributed over the PEs of a multi-processor system, then many could be searching for matches to a query literal. This requires informing the PEs of the search literal and organizing the results unless each PE could continue independently with the goal list.

3.1.4. Stream Parallelism

There are at least two senses in which the phrase *stream-parallelism* is used. The most often-cited definition is the one given by Conery and Kibler [CK81] and refers to the processing of structured (recursive functional) terms. If two literals, in a producer-consumer relationship, share a structured term, the consumer can begin processing the term before the producer has completely instantiated it. It can be said that stream-parallelism is the pipelining of structured data [BORG84]. For example, the consumer can start processing the head of a list before the producer has completely instantiated the tail. The second sense of the phrase considers logic programs to be collections of "stream transducers" [LP84] or assertion groups, each with its own unification processor [NAKAG84]. Here, "goal objects" or literals "stream" between transducers or processors and pick-up instantiations for terms that are not necessarily structured. This in effect establishes a partial-order on the instantiations much like that suggested for AND-parallelism. The difference is that while two conjuncted literals may be simultaneously active, they are active on *different* branches of the search space [GREG86]. Hence, only the first definition identifies a different source of parallelism related to the operation of unification.

3.1.5. Instantiation Parallelism

This refers to the process of substituting terms for variables and is clearly dependent on the representation of literals. One variable could have many occurrences in a string representation. In a pointer-linked structure, however, each variable could have just one pointer defining its current instantiation. How either of these representations is distributed over a multiprocessor affects the available parallelism. Chu and Itano [CI84] discuss instantiation-parallelism in conjunction with *uninstantiation-parallelism* during backtracking.

3.2. Parallel Logic Languages

3.2.1. Parallel Extensions

Given the inherent parallelism in logic languages, it should not be surprising that many explicit notations are intended to *control* rather than *facilitate* it. Wise describes a language Epilog [Wise82] that assumes uncontrolled AND/OR-parallelism in the search space. He then introduces the CAND and COR operators that specify *sequential* execution of literals and together can implement *cut* that depends on pruning branches in a left-to-right traversal of the search space. To limit the amount of useless computation, a type of mode declarations are defined. The first is a *threshold* which simply states *how many* of a literal's variables must be bound before its parallel execution can be initiated. In a more general scheme, each variable can be tagged with '!', '?' or nothing to signify whether it must be free, bound or either (respectively) before execution can be initiated.

While these annotations help control parallelism, they don't really help with the kind of parallelism that is involved in *systems programming*, i.e., where parallelism is actually part of the problem domain. One way to approach this kind of parallelism is simply adding to a sequential logic language the same kind of communication and synchronization primitives that were added to imperative languages.

This is the case of Delta-Prolog [PMCA86]. Synchronous sending and receiving are denoted by ‘!’ and ‘?’, respectively, e.g., two literals L and L' are unified at *event* E , when one process executes $L!E$ and another process executes $L'?E$. The first process to execute suspends until the second process synchronizes much like in Hoare’s CSP. While each process is assumed to be sequential and free to backtrack, in the absence of *distributed* backtracking, communication between them must be committed. Hence, communication primitives have *event conditions*, e.g., C and C' as in $L!E : C$ and $L'?E : C'$, which must be satisfied before the event can actually take place; much like a guard in CSP.

3.2.2. Guarded Clause Languages

Delta-Prolog, however, has the drawbacks that the natures of the communicating processes are fixed and there is no facility for operating system functions like process creation. A class of logic languages based on stream-parallelism through shared variables is intended to fill this niche. These languages are based on the Relational Language of Clark and Gregory [CG81] that defines a clause as:

$$P \leftarrow G_1 \wedge \dots \wedge G_k \mid S_1 // \dots // S_p$$

where P and each G_i are atoms and each S_i is a *sequential component* that is a conjunction $A_1 \wedge \dots \wedge A_m$ of atoms. Clauses are selected for execution much like Dijkstra’s guarded commands [DIJK75]. The *guard sequences* $G_1 \wedge \dots \wedge G_k$ of all matching P are tested in parallel and the first to succeed is selected, eliminating all others. It is said that the *commit* operator ‘|’ improves on the sequential *cut* since it is symmetric: all alternative computations are excluded rather than just those ‘below’ the cut. Hence, search parallelism is allowed but not OR-parallelism. Once a guard is satisfied, the corresponding goal set $S_1 // \dots // S_p$ is executed where ‘//’ is logically a conjunction but denotes parallelism and can be thought of as a *fork* between the sequential components which are then concurrent processes

communicating via streams through their shared variables. Synchronization is accomplished by restricting how unification of the shared variables is done. There are several methods of doing this.

The Relational Language [CG81] and its successor PARLOG [CG86] both have static mode declarations for each predicate such that each variable is declared *input* or *output* by '?' or '^', respectively. When a process (the stream consumer) tries to unify one of its input variables with another variable, it is suspended until another process (the stream producer) instantiates its corresponding output variable with a non-variable term. Rather than having static mode declarations, Concurrent Prolog [SHAP83] allows any occurrence of a variable to be declared *read-only* by '?'. Thus, a shared variable is declared read-only for the stream consumer but not for the stream producer. This is more flexible but increases the complexity of the unification algorithm which can now encounter the read-only declaration on any term. In Guarded Horn Clauses (GHC) [UEDA85], there are no declarations. If, during evaluation, a guard tries to bind a variable in the invoking call, the guard is suspended.

While guarded-clause languages have received a great deal of interest, they are difficult to implement in their full generality. Parallel execution generates a tree of bindings which are difficult to maintain, especially in a distributed environment and especially for those bindings involved in guards. Hence, versions of some of these languages have been developed which are either *safe* or *flat*. A clause is safe if unification of a guard never instantiates a goal variable to a non-variable term. This makes the tree of bindings easy to maintain. A clause is flat if the guard is restricted to system predicates that are easy to evaluate. This prevents the tree of bindings from being generated in the first place. A language with either property is more feasible to implement. Further comparison of these languages is given by Takeuchi and Furukawa [TF86].

3.3. Parallel Execution Models

Aside from the explicit expression of parallelism in logic languages, a tremendous amount of work has been reported on implementing systems that actually use the implicit parallelism. We now survey some of this work.

3.3.1. Shared-Memory Parallel Binding Environments

The fundamental problem for a shared-memory parallel execution model is how to maintain the multiple bindings for variables that are shared between OR-parallel and AND-parallel branches. A variety of methods have been proposed.

Directory Trees. Ciepielewski and Haridi [CH84, CIEP84] developed directory trees where a directory is a set of pointers to the stack frames that define a process' binding environment. When a new process is started, it copies its parent's directory and initializes a new frame for the called clause. A frame can be shared (pointed to from a directory) when it contains no free variables. Frames with free variables are copied on a demand basis. When a process wants to bind a free variable, it must look-up the frame containing the variable by following ancestor directories and then copy the frame.

Hash Windows. This technique developed by Borgwardt [BORG84] is similar to directory trees but the ancestor environment is not entirely copied since, in general, a child only binds a few ancestor variables. When a variable is needed, its value is looked-up in the local frame. If it is not there, ancestor environments and hash windows are searched until it is found. If it is free, then the binding is recorded in the local hash window.

Binding Arrays. This is yet another similar technique proposed by Warren [WARR84]. When a sequential interpreter binds an ancestor variable, it makes an entry on the trail stack such that the binding can be undone on backtracking. This approach "turns the trail stack around". The binding is made on a *forward list*.

A set of forward lists comprises a process' binding array which is its non-sharable information.

Variable Importation/Exportation. This approach by Lindstrom [LIND84] extends the current frame to include a slot for unbound variables in the parent frame. The unbound parent variable is *imported* by making an entry in an *import vector* that associates the current frame slot with the parent slot. When the clause is finished, all imported variables must be *exported* to a new copy of the parent frame.

3.3.2. Message-Passing Models

Non-shared memory poses different problems. Since "remote memory references" are prohibitively expensive, the problem becomes what should be communicated between PEs in a message. The *kabu-wake* method by Yasuhara and Nitadori gives each new process its own copy of the entire stack and has it backtrack to the branch it is to explore, unbinding any shared-variables in the process. The cost of copying the stack is offset by the ease of variable look-up in a non-shared environment. The *closed environment* approach by Conery [CON87] closes the child environment relative to its parent which means that there are no pointers in a child frame that point outside the current environment, i.e., to another PEs memory. This is ensured by a closing algorithm that closes the child frame by possibly enlarging it for any free parent variables. Further comparison of these and the shared-memory methods can be found in [CRAM85] and [CON87].

A different message-passing approach is taken by Lindstrom and Panangaden [LP84]. They associate a *stream transducer* with each literal that are organized into an AND/OR tree that follows the syntactic structure of the program. An OR node is associated with the head of a clause and the clause body is composed of AND nodes chained together from left to right such that when the OR node receives a goal, it is sent through the chain of AND nodes and returns with a complete set

of solutions. When an AND node receives a goal, it is sent to the OR node of every unifiable clause head which return all possible solutions. As sub-solutions progress through the chain of AND nodes, inconsistent ones are discarded. This approach only provides OR-parallelism and not AND-parallelism since the AND nodes are simply chained from left to right. The Sync Model by Li and Martin [LM86] also organizes computation in an AND/OR tree but achieves AND-parallelism by allowing body literals to execute in parallel and using special *Sync* messages to form the proper Cartesian product of partial solutions.

3.3.3. Dataflow Models

Given the easily identified parallelism in logic languages, it is not surprising that the dataflow model of computation has been applied to it. Umeyama and Tamura [UT83] report such a model that represents logic programs as dataflow graphs using five different operators: (1) *unification*, (2) *copy*, (3) *merge*, (4) *entry* and (5) *return*. A *unification* unit unifies literal data on two matching input messages. A *copy* unit merely replicates a message onto many output edges while a *merge* unit serializes messages from many input edges onto one output edge. Of more interest are the *entry* and *return* units – they manage the entry and return from a clause as if it was a procedure. *Entry* and *return* units always occur as a pair and have a matching number of inputs and outputs that equals the number of terms in the clause head. The literals in the clause body (sub-procedures) are solved from left to right in keeping with the procedural reading. Thus, AND-parallelism is not attempted. In fact, only OR-parallelism between procedures is done. To manage concurrent procedure activation (which could be due to recursion or OR-parallelism), each message is tagged with a *context* which is saved in a special message field when a new activation and context are entered.

Halim and Watson [HW84, HALIM86] describe a very similar model using five operators and tagged messages in a procedural reading to provide OR-parallelism

but not AND-parallelism. The operators are (1) *unify*, (2) *activate*, (3) *extract-output*, (4) *call* and (5) *return*. *Unify* is the same. *Activate* and *extract-output* are similar to *copy* and *merge* in that they replicate a message for distribution to all applicable clauses and then merge the results. In addition, *activate* must build the output message's *goal literal* and *binding environment* from its inputs. *Call* and *return* are similar to *entry* and *return* except the saved context is sent on a separate message from *call* to *return* instead of being stored in special field of the current query message.

Ito and Masuda [IM84] describe a more ambitious model that includes AND-parallelism and stream parallelism. It has a large set of operators that includes (1) *unify*, (2) *substitute*, (3) *consistency check*, (4) *create stream* and (5) *append stream*. Each *unify* operator unifies one pair of arguments from two literals and a *consistency check* operator examines the results from all arguments which in turn controls *substitute* operators. AND-parallelism must be explicitly indicated by annotation. After discussing several cases of such AND-parallelism, Ito and Masuda say that some (unspecified) consistency check operator would be needed here also.

3.3.4. Broadcast Models

Broadcast models of execution have also been investigated. Taylor, et al. [TLMS84] use one, supervisory *control processor* which contains all clause bodies and many PEs over which all clause heads and facts are partitioned. Execution proceeds by the control processor broadcasting commands to all PEs. The current goal list is expanded by an entire level at a time by initiating unification in the PEs. Since different PEs may produce inconsistent bindings, a join operation is done. Hence, AND/OR and search parallelism are accomplished in a SIMD style.

Warren, et al. [WADK84], also have proposed a broadcast model where the program is partitioned over the PEs. All predicates are classified as *universal* or

distributed. Each PE has a complete definition of all universal predicates. These would include 'low-level' system predicates such as *member* or *append*. The clauses for a distributed predicate are distributed over all PEs such that each clause is defined in only one PE. These would include 'high-level' user predicates. Queries are organized into zero or more universal predicates followed by zero or more *chunks* which are processed from left to right. Each chunk is a distributed predicate followed by zero or more universal predicates. Clearly, each chunk can be processed by one PE. Computation proceeds by a PE broadcasting chunks over the network which are picked-up by PEs that have the requisite predicates.

3.3.5. Language-Directed

Some work has actually chosen Prolog as the machine language and exhibit fine-grained unification parallelism. Chu and Itano [CI84] detail a machine where searching, unification and instantiation occur in parallel. If backtracking occurs, parallel *uninstantiation* takes place. On another tack, Tick and Warren [TW84] convert Prolog programs into a sequence of instructions from five different classes: *control*, *get*, *put*, *unify* and *index* from the Warren Abstract Machine [WARR83]. These instructions are handled by parallel *execution units*, thus accomplishing unification parallelism [TICK84].

4. Semantic Networks

We now discuss the third and last subject area that defines the boundaries of BPEM.

4.1. A Knowledge Representation

The *Semantic Network* is a method of *representing knowledge*, of encoding what is known about a given subject, that is widely used in Artificial Intelligence

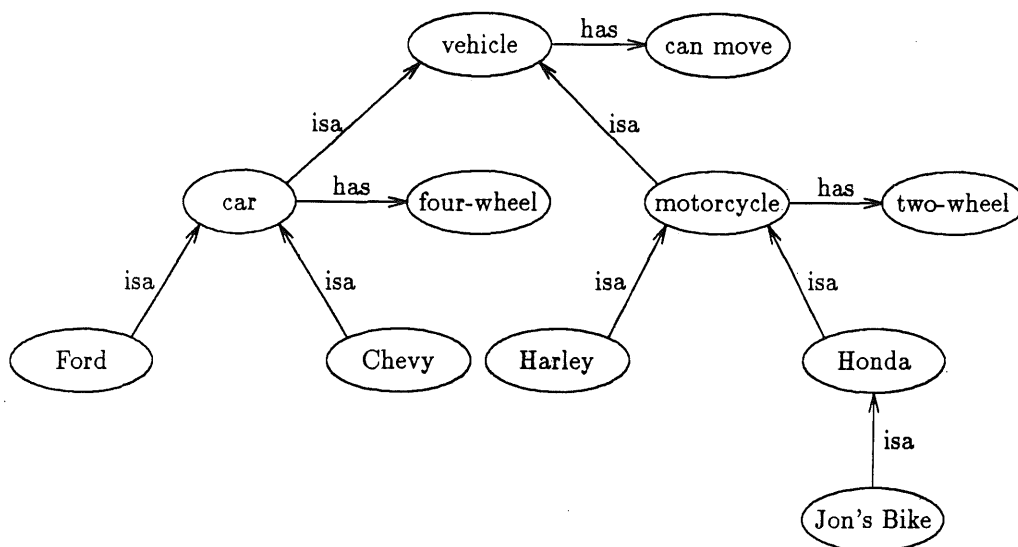


Figure 4

A Simple Vehicular Semantic Network

[FIN79, BF81, COM83, CM87]. The basis of semantic networks is that of an *entity-relationship hierarchy*, that is, a graphical representation consisting of vertices and edges. Broadly stated, vertices or *nodes* represent *concepts* and the edges or *links* represent the semantic *relationships* between them.

One of the most common uses of semantic networks is that of a taxonomic hierarchy. Concepts, possibly denoting objects, can be arranged in a hierarchy from most specific to most general and connected by *isa* edges (simply meaning “is a”) as shown in Figure 4. Among other things, this network states that a Ford is a car and a car is a vehicle. Similarly, a Harley-Davidson is a motorcycle and a motorcycle is also a vehicle. This hierarchy can also be extended to individuals as indicated by the fact that Jon’s bike is a Honda.

Another important aspect of semantic networks is that concepts can have *properties* or *attributes*. For example, vehicles “can move” and cars have “four wheels”. In addition, properties can be *inherited*, that is to say, a property can be considered as also belonging to all descendants in the hierarchy. Hence, we

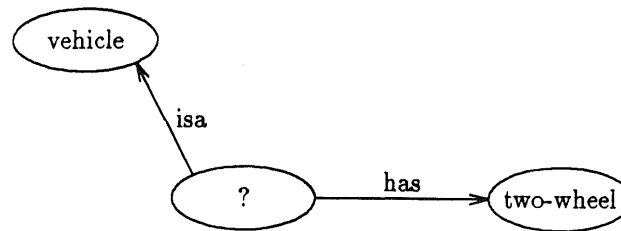


Figure 5

Network Fragment

can *infer* that Chevys have four wheels and can move. An obvious advantage of property inheritance is the efficiency of storage; a property that may have a broad applicability need only be stored once at the most general level.

What has been presented so far is only the network part of the entire semantic network formalism. The network part alone is said to comprise the *semantic memory*. In order to make use of a network, a set of *interpretive procedures* must be defined that operate on the network. One clear function for these procedures is to make *inferences*. As mentioned above, the possession of a property can be inferred by following the *isa* hierarchy. Inferences can also be drawn by matching network fragments. Assume we wish to find out what kind of vehicles have two wheels. This could be represented as in Figure 5. By matching “vehicles” and “two wheel”, we can conclude that motorcycle is the answer.

Another operation that can be done on semantic networks is *associative search*. One original intent of semantic networks was, in fact, an associative model of human memory. In this context, related ideas could be found by following edges like ripples spreading out in a pond. This is called *spreading activation* and the number of edges in a path between two concepts is the *semantic distance*. Some experimental evidence supports the idea that humans do take longer to verify statements involving a greater semantic distance, e.g., “a poodle is a dog” versus “a poodle is an animal”, but this is not strictly true [JLCH84]. In practice, spreading activation is constrained by techniques such as maximum distance, maximum

connectivity (fan-out) from a concept, and minimum strength of an association (weight of a link) [COBKJ87].

Semantic networks can be further organized by the use of *partitions* [HEN75, FIHE77, HEN79]. The nodes and edges of a network can be partitioned into *spaces*. Networks can then represent propositional formulas where each argument of any logical connective is a space. Spaces can also be used to define the scope of quantified variables and can be nested. The semantics of a semantic network can be more clearly defined by using *procedural attachment* [LM79]. This approach distributes the interpretive procedures over the entire network by associating with each node the procedures that are meaningful for it to perform.

Semantic networks certainly have their share of controversy concerning what their precise meaning is and what they are capable of. The question of meaning arises from what nodes and links are intended to represent. In Quillian's originating work on semantic memory [QUIL68], nodes represented natural language "word concepts". Groups of word concepts were linked together to form a *definition* and was considered to lie in a plane. Concepts within a definition could have their own definition via a link to another plane. These links defined relationships like subclass, subject/object, modification, conjunction and disjunction.

Over the years, however, many different things were represented with nodes and links resulting in many different interpretations. This is typified by the different shades of meaning given to the *isa* link as summarized by Brachman [BRACH83]. After noting that nodes have been used to represent concepts, sets, predicates, prototypes, descriptions, general terms, and individuals, among other things, Brachman makes a fundamental dichotomy between *generic* and *individual* interpretation of nodes. This leads to *generic/generic* relations and *generic/individual* relations. The *isa* link can be further characterized by the effects of node-meaning, e.g., whether the links are *sentence-forming* or *concept-forming*. In summary,

Brachman suggests that *isa* ought to be broken into sub-component parts which are *primitives* in the targeted representation problem.

This problem of determining what is to be represented was also examined by Woods [WOODS75]. One distinction he makes is between *extensional* and *intensional* knowledge which is the difference between denotation and meaning. For example, the extension of the concept *published* could simply be a predicate that is true or false for a given author. The intension of *published* is what it means to be published, i.e., writing, submission, review, acceptance, printing, distribution, tenure. This also illustrates that links in a network can be used not only as relations between concepts but also as predicates and that the two uses should be distinguished. Another distinction exists between *assertional* and *structural* links or properties. We can assert that George is x centimeters tall and assign any value to x without changing what it means to be tall. On the other hand, if every *buy* node has *agent*, *object* and *time* links associated with it, then these links structurally define what it means to buy something.

Brachman has also investigated "what knowledge is" with regard to semantic networks at a more global level [BRACH79]. Given the varied usages and meanings of the nodes and links of semantic networks, he identifies five levels at which networks can be understood:

- 1) Implementation,
- 2) Logical,
- 3) Epistemological,
- 4) Conceptual, and
- 5) Linguistic.

The implementation level is the simplest since nodes and links are just records and pointers with which to build a data structure. The logical level encodes logical connectives and predicates can be considered "foundational" due to the formal mathematical rigor which can be applied. The conceptual level is most commonly

associated with semantic networks since it deals with concepts and relations that can be limited to a well-defined set of “primitives”. On top is the linguistic level where nodes are language-specific word senses and topics like analogy, metaphor and anaphoric references must be resolved. In the middle is Brachman’s previously “missing” link; the epistemological level. This is based on formal knowledge-*structuring* primitives which can be applied to any concept domain. This is in fact Brachman’s main argument: the central problem in any representation scheme is finding the concepts and relations that are primitive at the desired level.

There is, of course, much more that could be said about the capabilities and inabilities of semantic networks. Johnson-Laird, et al. [JLCH84], lodge many complaints against semantic networks primarily at the linguistic level but what is interesting is their main thesis: all these problems stem from semantic networks being “only connections”, devoid of any *a priori* relation to the “real-world” which could give it true meaning. This is reminiscent of the formalist school of mathematics that holds that math is purely a formal symbol system with syntactic rules for manipulating marks on a page with no necessary relation to anything else whatsoever. This is in some sense true and emphasizes the difficulty in capturing *informal* ideas in a *formal* system which is at the heart of much of AI.

4.2. Applications

Despite any difficult subtleties, semantic networks have been applied in a variety of ways. We will review only a few.

Agarwal describes the use of semantic networks to solve problems in robotics [AGAR83]. The state of the robot’s world and a problem are represented as a network. Problems are solved by applying transformations that correspond to robot actions. Agarwal notes that this technique could be used to controlling multiple robots in parallel provided that interfering actions are prevented.

Ince reports a network for source code version control for major software development projects [INCE84]. A large piece of software is represented as a hierarchy of modules and submodules *including* different versions and subversions. The development history can also be represented along with the programmers responsible for maintaining each module. The language SOLO was developed to maintain the network and extract information from it.

Cohen and Kjeldsen describe the GRANT system that finds granting agencies for academic research proposals [COHKJ87]. The network here represents the interrelations between thousands of research topics in the health sciences. Each topic is connected to all granting agencies interested in that topic along with the type of interest, e.g., supply, educate, study, etc. Connections between agencies and proposals are found by spreading activation constrained by distance, fan-out and link strength. Performance data is given.

5. Combining All Three: Logic, Parallelism and Semantic Networks

We now discuss the intersection of all three subject areas. It has already been mentioned in the previous section that logic can be represented in semantic networks. This section establishes that connection more explicitly which carries with it the opportunities for parallelism. We can then state how BPEM is similar and dissimilar to research done separately in semantic networks and in parallel logic programming.

5.1. Representing Logic in Semantic Networks

First-order predicate logic can be represented in semantic networks given that the standard logical connectives, predicates and quantifiers can be represented. We now examine how these elements of logic can be expressed. Ultimately, however, we will be more interested in representing the clausal form of logic.

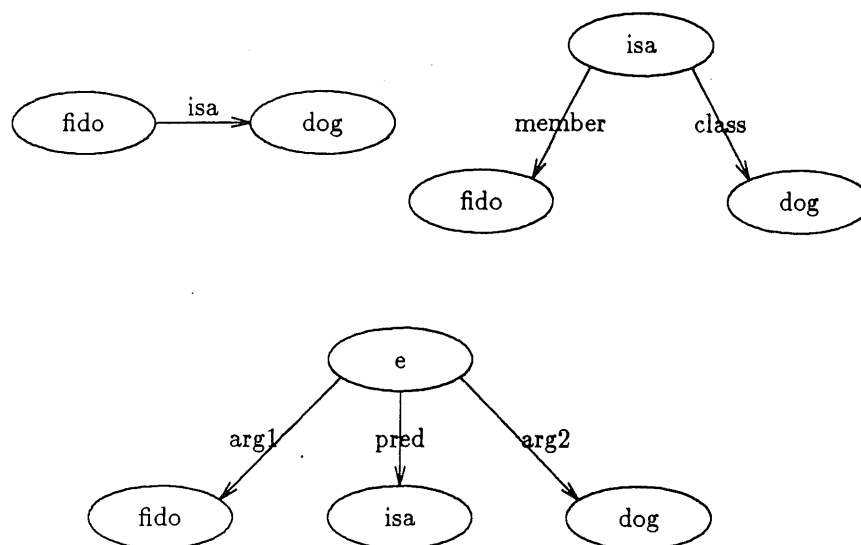


Figure 6

Representing Predicates

5.1.1. Predicates

Predicates can be represented as edges between the terms in several ways as illustrated by the literal $isa(fido, dog)$ as shown in Figure 6 [SHAP79]. Since this is a binary predicate, it can be expressed simply as an edge labeled with the predicate name connecting the two terms. The edge direction identifies the term order. (Conversely, any edge can be considered a binary predicate.) We can also represent the predicate name with a node connected to the arguments with edges labeled by the “role” the argument plays. Finally, we can represent the occurrence of the literal by introducing a new, unique node “e” connected to the predicate name and the arguments. This last case illustrates how any n -ary predicate can be represented as $n + 1$ binary predicates [DK79]. It is quite common to represent predicates in this manner [FIHE77, DK79, SHAP79, VK85]. This also illustrates the fact that any n -ary predicate can have a graphical representation. This is central to BPEM.

5.1.2. Functional Terms and Quantifiers

Functional terms are not always explicitly dealt with in semantic networks. This is in part because many networks are not built at the logical level but also because the way quantification is handled may not require the Skölemization of existential quantifiers. In some network formalisms, functional terms are just recorded as a term (node) regardless of their structure [DK79, VK85]. Because of their syntactic similarity with predicates, however, they can be represented in much the same way as predicates with edges connecting the term with each of its sub-terms [SGC79].

As noted before, quantification can be represented by network partitioning [HEN75, FIHE77, HEN79]. A partition is used to delimit the scope of an existentially quantified variable where each such variable node has an edge to the set in which the quantified element must exist. A universally quantified variable can be represented as an implication $\forall x \in X \rightarrow P(x)$. (See Section 5.1.3 for the representation of connectives.) Here, the antecedent essentially types the universally quantified variable and the consequent must be true for all values that satisfy the typing.

5.1.3. Connectives and Clauses

The usual logical connectives, including negation and implication, can also be represented by partitions. Each partition contains the terms in a conjunction or disjunction or the term to be negated or the antecedent and consequent of an implication. This association between the terms can also be made by a special node that identifies the connective, e.g., an *or*-node, and that has edges to each term involved [SGC79].

It is more common, however, to express formulae in *clausal* form. In this case, all antecedents are implicitly *or*-ed and all consequents are implicitly *and*-ed. Hence, only a distinction between antecedent and consequent needs to be made. This is easily done by typing or coloring the edges [DK79, SHAP79, VK85].

5.2. Inferencing

Given that semantic networks can be viewed as a syntactic variant of logic, it must be possible to perform logical operations such as deduction. Since the network formalisms are not all the same variant, how inferencing "looks" is different. Common among them all, however, is the matching between conditions and conclusions in order to draw an inference.

5.2.1. Matching

Inferencing using partitions is done by extending the notion of spaces to that of a KVISTA, representing a knowledge vista, and a QVISTA, representing a query vista [FIHE77]. The QVISTA may contain variables and the desired answers, the bindings, must be found by matching the two vistas and bindings the variables to elements in the KVISTA. Actually finding matches is a complicated matter embodied in a system called SNIFFER. SNIFFER builds a tree of alternative solutions, assigns them a priority level and uses "strategy selectors" which invoke "expert" binding tasks that find matches in specialized classes of relationships.

Shapiro [SHAP79] describes the network representations for four deduction rules that are more organized yet nonetheless non-standard. These are:

- 1) \vee -entailment: $\{A_1, \dots, A_n\} \vee \rightarrow \{C_1, \dots, C_m\}$ is true if each A_i , $1 \leq i \leq n$, entails each C_j , $1 \leq j \leq m$.
- 2) \wedge -entailment: $\{A_1, \dots, A_n\} \wedge \rightarrow \{C_1, \dots, C_m\}$ is true if each C_j , $1 \leq j \leq m$, is entailed by the conjunction of the A_i , $1 \leq i \leq n$.
- 3) AND-OR: $\bowtie_i^j \{P_1, \dots, P_n\}$ is true if at least i and at most j of the P are true.
- 4) THRESH: $\Theta_i \{P_1, \dots, P_n\}$ is true if fewer than i of the P are true or they all are.

While these deduction rules may be useful in representing deductive situations occurring in natural language, any process that must match these structures to draw an inference must reflect their increased complexity. Rather than having

a straight-forward method of inferring the desired goal, some strategy heuristics must be used.

The use of a clausal form of logic adds tremendous structure to the inferencing process and generally follows the constraints for theorem-proving in logic described earlier in this chapter. Given that the conditions and conclusions for clauses are identified, inferences can be drawn by successive steps of resolution. Deliyanni and Kowalski [DK79] use two different edge types for conditions and conclusions and allow constants to be shared between clauses since clauses are (pictorially) partitioned. A query graph is added and then unification and resolution are used to deduce the empty graph. A clausal representation is also used by Vagin and Kiknadze [VK85]. These network formalisms are, in fact, the closest syntactic variants of logic.

5.2.2. Efficiency

Clausal form still has the same efficiency concerns as discussed earlier but there are several techniques that can be used. The first is simply the use of Horn clauses which simplifies the search and computation rules. This also allows a procedural reading of the clauses.

Another approach is to use a *many-sorted* logic. It is not uncommon that a variable can only take values from a specific set. For example,

$$X = \textit{brando}, \textit{football_player}(X)$$

can never succeed since X should only be bound to football players which Marlon Brando is not. Hence, unification can be made more efficient by essentially *typing* the variables which identifies which *sort* they can take values from. This is used in [McSM79] and [VK85]. Of course, the unification algorithm must be modified to use the sort information. In [McSM79], the sort information is stored in an acyclic hierarchy which facilitates finding the intersection of sorts during unification.

Efficiency is also improved by the inherent property of *indexing* that semantic networks have. By their graphical nature, terms that occur in more than one clause can be represented by the same node. Thus, all other relations (edges) that term has are immediately available without searching. For example, an ancestor hierarchy may have thousands of *mother* facts recorded but *mother(joe,X)* will have only one *mother* edge that is relevant. This can greatly decrease the amount of time required to search for all possible matches. *This type of indexing occurs naturally in BPEM.*

There is another way of improving efficiency that is unique to BPEM. Other models of parallel logic programming encompass logic programming in general. This requires high overhead in maintaining parallel binding environments, as discussed in Section 3.3, especially in the case of building arbitrary functional terms which can require remote data lookups to accomplish unification. BPEM can accomplish the processing necessary for semantic networks without this overhead by restricting how functional terms can be used. This means that BPEM will not be a general programming system but it will be a completely asynchronous, distributed inference engine.

5.3. Parallelism

Many of the semantic network formalisms that have been described here lacked parallel processing capability. BPEM, however, provides a massively-parallel context for semantic networks and knowledge bases. It does this using logic wherein parallelism is easily identified. The use of logic also provides a clearly understood programming paradigm that is lacking in many semantic network models. The next chapter draws on the network formalism to develop BPEM as a distributed, massively-parallel execution model for a subset of logic.

CHAPTER 3

The Binary Predicate Execution Model

1. Basic Concepts

This chapter introduces the Binary Predicate Execution Model (BPEM). It is based on predicate logic and resolution as discussed in the last chapter with the distinction that most predicates must be binary or binarized. Specifically, all assertions and all query literals that would match an assertion must be binary. The motivation for this is that binary predicates have a simple graphical interpretation: the terms are the endpoints of a directed edge that is labeled with the predicate name. If this is done for assertions and queries, then resolution becomes a process of graph matching. If we consider the assertion terms to be active processing elements (PEs) and the assertion edges to be communication channels, then this graph matching can be accomplished by exchanging messages about the portion of the query that remains to be matched.

This graphical interpretation is the basis for BPEM's distributed, massively-parallel nature. This does, however, impose certain restrictions for efficient execution. Hence, this chapter will concomitantly introduce Restricted Binary Logic (RBL), the subset of first-order predicate logic that is used by BPEM. We begin by reviewing the binarization of predicates, their graphical interpretation and resolution as a process of graph matching. Subsequently, clause invocation, functional terms, attributes and the control of parallelism will be discussed. We will conclude with a comparison of other work. Much of this material also appears in [BL87].

1.1. Binarization of Predicates

As noted in the last chapter, any n -ary literals can be represented as $n + 1$ binary literals where the predicate name p and each argument are associated via a new constant symbol e that denotes a specific literal and new predicates that define the “roles” that each argument played. As special cases, if $n = 2$, the transformation is identity and if $n = 1$, only one binary literal need be introduced. The important thing is that the semantics of the n -ary literal are syntactically preserved in the binary literals.

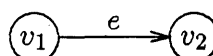
Also noted was that $n + 1$ binary literals can be reduced to only n . Since we are dealing with first-order logic, the predicate name is a constant. In this case, the new constant symbol that denotes the literal can be made a tuple of a new constant symbol and the predicate name, $\langle e, p \rangle$, denoting the unique occurrence of this literal. For example, $p(x_1, x_2, \dots, x_n)$ can be written as:

$$\begin{aligned} & \text{first_arg}(\langle e, p \rangle, x_1). \\ & \text{second_arg}(\langle e, p \rangle, x_2). \\ & \dots \\ & \text{n-th_arg}(\langle e, p \rangle, x_n). \end{aligned}$$

This reduction in the number of literals is paid for by the increased cost of unification with a tuple. An advantage, however, is that the predicate name is recorded with each literal. As we shall see later in this chapter, clause heads will not be binarized in order to handle conflicting data dependencies between clauses.

1.2. The Assertion Graph and Query Templates

Since we are now only dealing with binary literals, we can represent logic programs as graphs. To begin, any literal $e(v_1, v_2)$ may be transformed into a directed edge of the form:



The arrowhead records the order in which the terms of the literal were given. This information must be preserved when the literal represents an asymmetric relation.

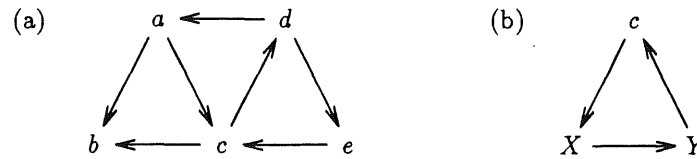


Figure 7

An Assertion Graph and Query Template

As will be discussed later, the arrowheads do not prescribe the direction in which messages may flow through the graph. *Note:* Since an edge is just another way of representing the same information contained in a literal, we will use the expressions ‘literal’ and ‘edge’ as synonyms. Similarly, the expressions ‘term’ and ‘vertex’ will refer to the same concept.

With this graphical interpretation, the set of all assertions containing only ground terms, representing the set of explicit facts, form the *Assertion Graph* (AG). This graph is also referred to as the *extensional database* [GAMI78]. Note that multiple occurrences of any ground term are mapped onto the same vertex of the assertion graph. The AG is a *task interaction graph* which implies that each vertex is an active element capable of receiving, processing, and emitting messages traveling asynchronously along the graph edges.

In a way similar to the AG, any query body and all clause bodies may be viewed as a graph. These will be referred to as *query templates* (QTs). We note that this set of clauses is also referred to as the *intensional database* [GAMI78]. While all AG vertices must be constants, QT vertices are allowed to be constants or variables.

Example: Figure 7a shows the assertion graph for the facts

$$\begin{array}{cccc}
 p(a, b). & p(c, b). & p(d, a). & p(e, c). \\
 p(a, c). & p(c, d). & p(d, e). &
 \end{array}$$

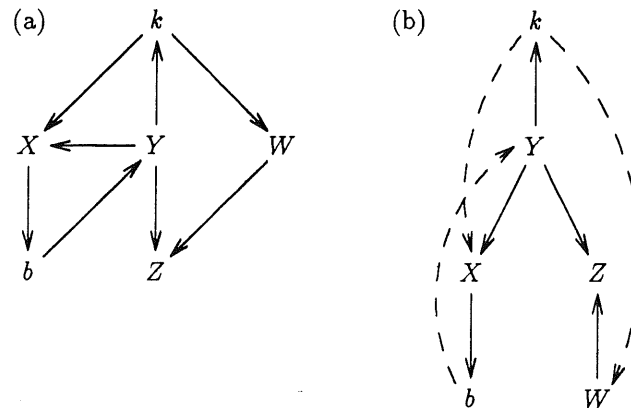


Figure 8

A Query Template and One Possible Spanning Tree

Figure 7b shows the QT for

$$\leftarrow p(c, X), p(X, Y), p(Y, c).$$

1.3. Graph Matching as Resolution

With this graphical interpretation of facts and clauses, we can show how resolution can be interpreted as graph matching. One step of resolution corresponds to matching a QT edge to an AG edge. This requires that both pairs of endpoints match, i.e., are unifiable, and that the edge label and direction match. A pair of endpoints unify if (1) they are the same constant, or (2) a variable (from the QT) is *bound* to the AG constant. Finding a solution corresponds to finding a match for each edge and vertex in the QT. Finding all solutions means finding all matches.

For example, the AG and QT in Figure 7 have two solutions given that c must be matched between the two graphs: $\{d/X, a/Y\}$ and $\{d/X, e/Y\}$. All other bindings for X and Y are incorrect since the edge directions do not match.

Since a QT represents a conjunction of literals, opportunities for AND-parallelism may exist. As discussed in the previous chapter, consistent bindings for any variables shared between AND-parallel literals can be obtained by performing an equi-join operation after the bindings have been produced or by defining

producer-consumer relationships to prevent inconsistent bindings from being produced. The extra work of an equi-join can be avoided by using *depth-first spanning trees* to define the order of binding, i.e., the producer-consumer relationships, among the variables and also to identify independent sub-trees in the QT that can be done AND-parallel.

Figure 8a shows a QT and Figure 8b shows one possible spanning tree for it. Any depth-first spanning tree partitions a graph into *tree-edges* (solid lines) and *back-edges* (dashed arcs). Starting at the root k , matching can proceed along all descendent tree-edges in parallel until the leaves are reached. Of course, a spanning tree may be rooted at any vertex and any vertex may be the root of many different spanning trees. In general, the top-level query is rooted at some constant since this constant must match in the AG. Each clause body spanning tree is rooted depending on which variable is bound first in the calling literal. Note that each root may have many spanning trees with differing execution costs. This is discussed in a later section. Also note that the spanning tree is defined *as if* the graph is undirected. This is because the edge direction does not affect the order of binding. It makes no difference that binding is conceptually done from head to tail or tail to head just so long as the edge directions match.

In a spanning tree, any vertex at depth i is the *generator* for all descendants at depth $i + 1$. For any vertex v with a back-edge to some ancestor w , it is easy to show that w is always bound before v since $depth(v) > depth(w)$. Hence, the matching of back-edges can be done from the deeper vertex, guaranteeing that both vertices are bound at the time of matching, thus avoiding inconsistent bindings. Since there are no cross-edges, there are no dependencies between branches. Thus, all descendent tree-edges from a vertex represent independent sub-queries that can be processed in parallel.

This tree structure can be easily represented using a nested syntax. Hence, the query from Figure 7b can be written as

$$\leftarrow p(c, X)[p(X, Y)[p(Y, c)]]$$

The query from Figure 8 can be written as

$$\begin{aligned} \leftarrow & p(Y, k)[\\ & p(Y, X)[\\ & \quad p(k, X), \\ & \quad p(X, b)[\\ & \quad \quad p(b, Y) \\ & \quad] \\ &], \\ & p(Y, Z)[\\ & \quad p(W, Z)[\\ & \quad \quad p(k, W) \\ & \quad] \\ &] \end{aligned}$$

assuming that all predicates are labeled p . Note again that the order of the terms in each literal corresponds to the direction of the edge but this does not affect the order of matching.

Given that the AG is considered a task interaction graph and that a QT is represented as a depth-first spanning tree with back-edges, matches between them may be found in the following manner.

- 1) Select a QT vertex v_q as the root. It is best if v_q is a constant that occurs in the AG since all solutions would have to match here. If the QT only contains variables, then any v_q could potentially match any AG vertex. In this case, however, only those AG vertices with the same incident edges as v_q must be attempted.
- 2) Place a description of the QT on a *query message* and "inject" it into the AG at the matching AG vertex v_a .

- 3) Unify v_q with v_a .
 - A) If this fails, then the (sub)query fails; send a *failure reply message* to the sender of this message.
 - B) If this succeeds, then
 - a) If there is more to match, then send new query messages along all AG edges incident on v_a that match descendent tree-edges and ascendant back-edges from v_q . Repeat Step 3 with the vertices on the other ends of these edges as the new v_a and v_q .
 - b) If there is no more to match (at a leaf), then send a *success reply message* to the sender of this message with any binding that was made.
- 4) If at least one success message is received for each descendent tree-edge and ascendant back-edge, then send a new success message to the sender of the original message. Otherwise, send a failure message.
- 5) When the initial v_a has received replies for all initial query messages, then all solutions have been found.

1.4. An Example

The AG and QT of Figure 7 would produce the following sequence of events:

- 1) Inject QT at c .
- 2) c finds matches for the first edge and sends query messages to b and d .
- 3a) b binds X but fails since it can't match the next QT edge; it sends a failure message back to c .
- 3b) d binds X , matches the second edge and sends query messages to a and e .
- 4a) c receives the failure message from b .
- 4b) a binds Y , matches the last edge and sends query messages to b and c .
- 4c) e binds Y , matches the last edge and sends a query message to c .
- 5a) b doesn't match c ; sends a failure message back to a .
- 5b) c matches itself and sends success messages back to both a and e .
- 6a) a receives a failure from b and a success from c ; it sends a success message with $\{a/Y\}$ back to d .
- 6b) e receives a success from c and sends $\{e/Y\}$ back to d .
- 7) d receives success messages from a and e ; it sends $\{d/X, a/Y\}$ and $\{d/X, e/Y\}$ back to c .
- 8) c receives final solutions.

This is a simplified, conceptual description of how the model accomplishes graph matching by message-passing. Actual implementation techniques are significantly different and are covered in Chapter 5.

2. Clause Invocation

In conventional logic implementations, no distinction is made between unification with a clause whose body is empty or not. The only difference is that *after* successful unification, the length of the goal list either decreases or it doesn't. BPEM, however, handles the two cases very differently. As described in the previous section, a step of resolution involving unification with an assertion that occurs in the AG corresponds to messages sent between PEs. Clause heads, however, do not occur in the AG. Hence, clause head unification must be done within one PE. Because of this, and because of potential data dependencies when invoking binarized n -ary clauses, every predicate name must be associated with either assertions or clause heads, not both, thus making it easy to determine when clause invocation is required. We will first introduce clause invocation in the simplest case when the clause head is already binary. We then give the general case for clause invocation that handles the invocation of binarized n -ary clauses.

2.1. Invoking Binary Clause Heads

As discussed earlier, a clause body can be viewed as a graph and, hence, transformed into a spanning tree in the same manner as an initial query. The edge representing a binary clause head can be replaced by the tree representing its body where the end-points (variables) of the head are unified with the vertices (constants or variables) in the query tree. As an example, Figure 9a shows the query

$$\leftarrow p(a, B), p(a, C), p(B, D), q(B, E), p(E, F), p(E, G).$$

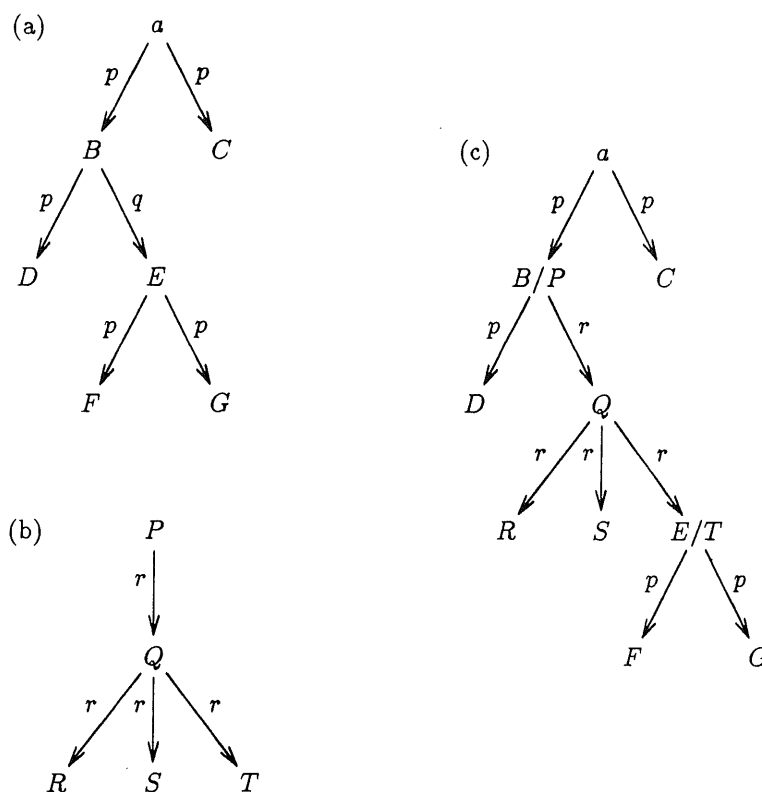


Figure 9

A Clause Invocation

Figure 9b shows the clause body for

$$q(P, T) \leftarrow r(P, Q), r(Q, R), r(Q, S), r(Q, T).$$

Assume that p matches only assertions but q matches the head for the clause shown in Figure 9b. The query tree that results from the invocation is shown in Figure 9c. The edge q in Figure 9a has been replaced by the tree in Figure 9b where B and P have been unified and E and T have been unified.

Note that any back-edges in the query template would just be “carried-along” after the invocation. If there had been a back-edge from F to a , then this same back-edge would exist after invocation even though the tree path between F and a may have changed. Also, if more than one clause head for predicate q could be unified, then an equal number of tree substitutions would occur, increasing

the OR-parallelism. It is important to note that, in general, the variable names (identifiers) in the resultant query must be modified to avoid identifier collision. If D appeared in the clause body for predicate q in Figure 9b, then identifier collision would occur after invocation.

This scheme handles recursion given that identifier-collision is avoided. This also means that non-terminating computations are possible. A query template could in theory become infinite due to a recursively-defined predicate. Matching such an infinite tree to the AG constitutes a non-terminating computation. Since any AG is finite in size, the only way such a match could occur is if the AG is *cyclic*. If the AG was not cyclic then only part of the query tree could be matched and only those solutions found so far (if any) would be returned. Consider the transitive *brother* relation:

$$\text{brother}(sam, bob). \quad \text{brother}(bob, joe). \quad \text{brother}(joe, sam).$$

If we define

$$\text{find_brother}(X, Y) \leftarrow \text{brother}(X, Y).$$

$$\text{find_brother}(X, Z) \leftarrow \text{brother}(X, Y), \text{find_brother}(Y, Z).$$

and then ask $\leftarrow \text{find_brother}(sam, X)$, an infinite number of answers will be generated even though only three will be unique. If one *brother* fact is removed then the AG becomes acyclic. Now, even though the query tree is potentially infinite, only a finite sub-tree can be matched and only a finite number of answers produced. Unfortunately, being able to detect cyclic matching beforehand is tantamount to solving the halting problem. Practical methods for dealing with this dynamically are discussed in [WoB186].

2.2. Invoking Binarized n -ary Clause Heads

In the previous subsections, depth-first spanning trees were used to handle the *data dependencies* within the goal list. If any *cyclic dependencies* appear in the goal list (ignoring edge directions), one edge is guaranteed to become a back-edge.

$$\leftarrow p(a, B, C), s(B, C).$$

$$p(X, Y, Z) \leftarrow q(X, Z), r(Z, Y).$$

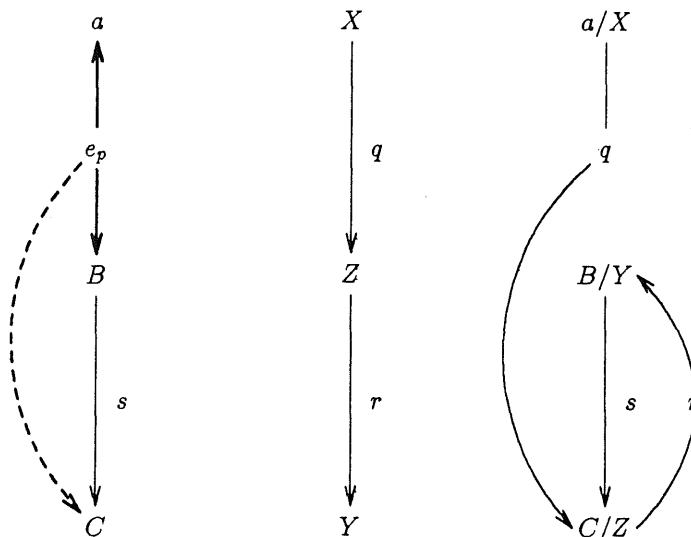


Figure 10

Clause Invocation with Binding Order Conflict

With this property, matching can proceed down the tree-edges without any need of synchronization.

Now consider what happens when a binarized n -ary literal in a query template is unified with a clause head as shown in Figure 10. The query template appears on the left and the clause body template for $p(X, Y, Z)$ appears center. The query template has the binding order $a \mapsto B \mapsto C$ while the clause body has the order $X \mapsto Z \mapsto Y$. Vertex e_p and its incident edges (all shown in bold) are to be replaced by the clause graph. The resultant graph with the appropriate unifications appears on the right. Note that both paths between B and C consist of tree-edges that expect the binding of B and C to occur in the opposite and *conflicting* order. The clause body will bind C and then B . After completing the the clause, the query template expects to use its binding for B to generate bindings for C which has already been bound. Hence, clause invocation superimposes the data dependencies

$\leftarrow p(x, A, B), q(A, B, C, D), r(X, C, D).$

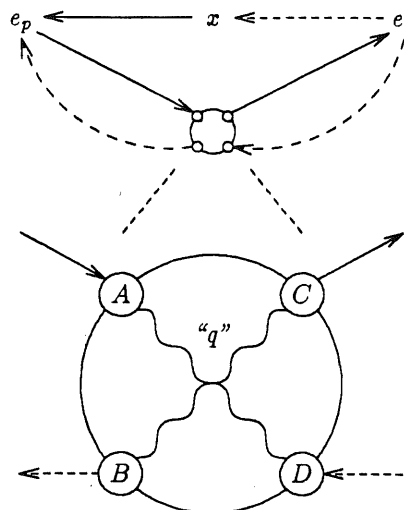


Figure 11

Invoking Literal as One Vertex

of the invoked 'child' clause on those of the invoking 'parent' template which may involve conflicting orders of binding.

One possible solution would be to redefine the spanning tree for the query template on the fly to agree with the binding order required by the child. Unfortunately, this is not practical. The PE handling the invocation would have to notify every other PE that could possibly handle B and C of the redefinition which is effective only for this clause. If other clauses matched $p(a, B, C)$, a redefinition could be necessary for each one. Thus, the redefined spanning tree would have to be carried on every subsequent message from a drastically increasing the communication and message processing time.

The solution used here rests on the observation that invoking literals are never matched to the AG. In Figure 10, vertex e_p and its incident edges are never matched. Since this is the case, it does not make sense to define a spanning tree, *which defines the propagation of messages through the AG during computation,*

using the sub-graph of the literal. Thus, every literal that can match a clause will be represented by *one vertex* in a clause body or query template.

Consider the example given in Figure 11. The query at the top of the figure has a spanning tree as shown immediately below it. This tree is rooted at x with solid tree-edges and dashed back-edges. Given that the literals p and r match assertions, they appear in the graph template in their binarized form and are used as part of the spanning tree. Given that the literal q matches a clause head, it must appear as one “macro” vertex in the spanning tree. The arguments of q appear on the boundary of this “macro” vertex as shown at the bottom of the figure. In the most general case, there is one incoming tree-edge. There can, however, be multiple outgoing tree-edges and multiple incoming and outgoing back-edges. Since binding proceeds from the root to the leaves of the spanning tree, binding information first arrives at A which then initiates the clause invocation. Any invoked q clause template must provide some path between A and the other arguments such that back-edges from B to some ancestor of A may be checked and that matching of the spanning tree may continue from C which may eventually check back-edges against D .

In a sense, the macro vertex acts as an interface between the parent template and the invoked child template. The parent’s spanning tree is defined as if all arguments of a clause head are bound at once. In execution, however, the binding order is defined by the child’s spanning tree. This requires two properties that have not been heretofore explicitly stated and also introduces some synchronization concerns. We first present the properties and then address the synchronization problem.

- 1.) *Every predicate name must be associated with either assertions or clause heads; not both.* Clauses like

$$p(a, b). \\ p(X, Y) \leftarrow q(X, Z), p(Z, Y).$$

are not allowed since a literal cannot be represented as both a graph with edges and a single vertex at the same time which would be necessary if p occurred as both a fact and a clause head. This is remedied by replacing every non-fact occurrence by a new predicate p' and adding new clauses to the fact occurrences:

$$\begin{aligned} p(a, b). \\ p'(a, b) &\leftarrow p(a, b). \\ p'(X, Y) &\leftarrow q(X, Z), p'(Z, Y). \end{aligned}$$

2.) *Every clause graph must be connected.* Clauses like

$$p(W, X, Y, Z) \leftarrow q(W, X), r(Y, Z).$$

are not allowed since, in general, the PE handling the invocation (as A in Figure 11) will eventually need a path to the PEs handling every other variable occurring in the head. This is remedied by creating a new clause for each connected component. Thus, a clause

$$p(\bar{x}, \bar{y}) \leftarrow q_1(\bar{x}), \dots, q_n(\bar{x}), r_1(\bar{y}), \dots, r_m(\bar{y}).$$

with two disjoint, connected components defined over the variables \bar{x} and \bar{y} , is replaced by

$$\begin{aligned} p_1(\bar{x}) &\leftarrow q_1(\bar{x}), \dots, q_n(\bar{x}). \\ p_2(\bar{y}) &\leftarrow r_1(\bar{y}), \dots, r_m(\bar{y}). \end{aligned}$$

and every reference to the head

$$\dots, p(\bar{x}, \bar{y}), \dots$$

is replaced with

$$\dots, p_1(\bar{x}), p_2(\bar{y}), \dots$$

Of course, changing one clause may require changing others and two clauses for p may not have the same components defined over the same variables, thus requiring more new clauses. This is a finite process, however, since any program has a finite number of clauses and variables.

To completely understand how the macro vertex functions, we consider each case of how information flows between A , B , C , and D and how to accomplish synchronization when necessary. The process of invocation is initiated at A which may receive binding information along the tree-edge.

- Every out-going back-edge, as from B , checks some ancestor of A in the parent template. Hence, all necessary binding information entered with the message to A . This information must be passed along some path in the child clause to B such the back-edge can be checked.
- The flow of data between C and D depends on the fact that every back-edge to D is descendent from one specific C . (If this was not the case, then the back-edge to D must come from some ancestor of A , in which case it can't be a back-edge, or from some aunt or uncle of A , in which case it would be a cross-edge, which can't exist in a depth-first spanning tree. The flow of data from A to D and its specific C depends on their binding order in the child template:
 - If the child clause has the binding order $A \mapsto D \mapsto C$ then there is no problem. All PEs handling D s pass their binding information to their respective C s such that some descendent in the parent template can check for the back-edge.
 - If the child binding order is $A \mapsto C \mapsto D$ then each PE binding a C initiates the sub-query in the child clause that will bind D but *suspends* the outgoing tree-edge in the parent graph. If the sub-query to D succeeds and returns bindings, then the sub-query along the parent tree-edge is initiated with D 's binding information such that the back-edge to D can be checked by a descendent in the parent template.
 - If the binding order is $A \mapsto X \mapsto C$ and $X \mapsto D$ where X is the most recent ancestor of both C and D in the child clause, then D returns its binding to X and C returns its *network address*. X then forms the cross-product of each binding and address and sends each cross-product element to the appropriate C which then initiates the parent sub-query with the appropriate binding of D .

In the best case, when there are no conflicting binding orders, i.e., data dependencies, this scheme does not change the behavior of the execution model. When conflicts do occur, some synchronization must be done between the PEs binding a D and its C to resolve the conflict. In the worst case, when the most recent ancestor X is identical with A , invoking a clause becomes like a procedure call. The child clause must execute completely and all head variables must be bound before any processing in the parent graph continues. This limits parallelism, however, only at this particular interface. All parallelism initiated before arriving at q continues and all child clauses can execute in parallel.

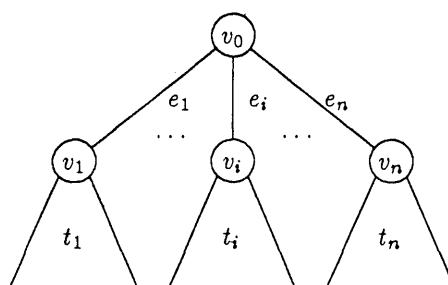


Figure 12

General Query Tree

3. Spanning Tree Cost

In the preceding query template examples, there was usually only one convenient root for the template spanning tree. In general, however, there could be many possible roots (and injection points) from which the same solutions could be found. The difference between spanning trees is the amount of computation required to arrive at the solution bindings. This subsection develops the idea of *spanning tree cost* to compare otherwise equivalent spanning trees.

To satisfy any literal (edge) in the graph requires finding all possible matches or bindings for its end-points. The computational resources required for this will be called the *cost*. Not surprisingly, the cost of different literals will vary since they will have a different number of possible solutions. Furthermore, the cost of one particular literal can also vary according to the direction in which matching is being done; from tail to head or head to tail. Thus, the aggregate cost between entire spanning trees can vary greatly even though they cover the same connected graph and are used to compute the same solutions.

Cost is clearly proportional to the number of messages that get sent. In the model as it has been presented so far, there is one reply message for each activation message. So the cost of a tree, or of an entire computation, can be characterized by the number of activation messages. We can define the cost of a spanning tree

by the following recursive equation (with reference to Figure 12) that starts with the tree-edges incident on the root:

$$W_{tree} = \sum_{1 \leq i \leq n} (W_{e_i} + K_{v_i} \cdot W_{t_i}) \quad (1)$$

where,

- W_{tree} = the cost of the entire (sub)tree,
- W_{e_i} = the cost of this particular edge, regardless of whether it is an assertion edge or a clause head,
- K_{v_i} = the number of bindings for v_i that require the sub-tree to be satisfied, and
- W_{t_i} = the cost of the sub-tree.

Intuitively, the cost of the entire tree is calculated recursively by adding the costs of satisfying each immediate tree-edge, e_i , and the corresponding sub-tree, t_i . Satisfying e_i produces zero or more bindings for v_i which requires an equal number of solutions for the sub-tree t_i . If e_i matches an assertion edge, then W_{e_i} is the number of activation messages. In this case, v_0 is bound and each activation message sent by v_0 results in a binding for v_i , so $K_{v_i} = W_{e_i}$. If e_i is a clause head, then W_{e_i} is calculated by applying equation (1) to the clause tree. The value of K_{v_i} , however, depends on the location of the second argument within the tree; it is the product of the number of solutions on the path from v_0 to v_i . Hence, for every vertex w on the path $v_0 \rightarrow v_i$ excluding v_0 , the definition for K_{v_i} can be written as:

$$K_{v_i} = \begin{cases} W_{e_i}, & \text{if } e_i \text{ is an assertion edge;} \\ \prod_{v_0 \rightarrow w \rightarrow v_i} K_w, & \text{if } e_i \text{ is a clause edge.} \end{cases}$$

In both cases, if e_i is the final edge in a tree branch, then t_i is empty with a cost of zero.

Hence, it is possible to compare the cost of different spanning trees and choose the most efficient for execution given that the cost of each assertion tree-edge is known. This cost is determined by the occurrences of the assertion edge label in the assertion graph. In general, however, an assertion edge in a query will not

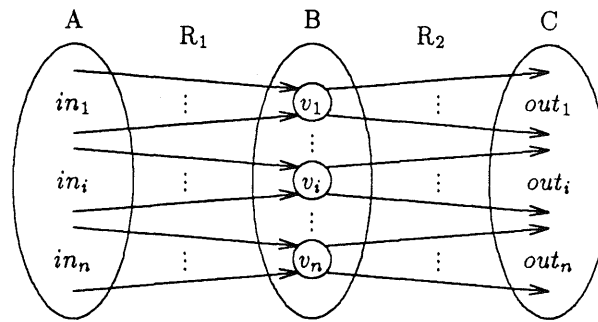


Figure 13

Domain Element Fan-in/Fan-out

necessarily cause an activation message to be sent on every matching edge. An unsuccessful query may not send any messages. A query may be successful having only sent one message for each edge in the query. Partial solutions that eventually fail may generate many messages before failure is established and so on.

To approach this problem, a technique for *estimating* the *average* number of activation messages has been developed that is based on a weighted average of edges incident on vertices in the assertion graph. Consider Figure 13. It shows three sets A, B, and C that are the domains and ranges of two relations (sets of assertions) R_1 and R_2 where B is shared. If we want to determine how many messages (on the average) flow out of domain B for predicate R_2 , then what we need to know is *how many out-going messages (on the average) are produced by each in-coming message* for some given predicate R_1 with domain A. This requires knowing the in-degree and out-degree of every vertex in any domain B that is shared between any two predicates R_1 and R_2 such that the average number of messages for R_2 can be calculated using the following *weighted average*:

$$W_{e_i} = \frac{\sum_{1 \leq i \leq n} in_i \cdot out_i}{\sum_{1 \leq i \leq n} in_i}$$

Here, R_2 is taken to be an assertion edge e_i in the original statement of the cost function, equation (1), if only one argument is instantiated. If both arguments are

instantiated, then $W_{e_i} = r_2/(b \cdot c)$, which is the probability that is met by every message from A that an edge exists between any two specific elements in B and C.

While this cost estimate is fairly accurate, it does require computing the weighted average for every pair of edges in a query. While these averages could be pre-computed for an entire AG, they still represent a significant cost themselves. In order to be useful, the computation of the cost function should cost significantly less than the work avoided by a more efficient spanning tree. In situations where large computations on large databases are contemplated, the investment may well be worth it. In smaller tasks, however, any tree may do since the choice of tree affects only the efficiency of an implementation and not the correctness of the model. Simpler but less accurate estimates are investigated in [LEE85].

4. Functional Terms and Variables in the Assertion Graph

As introduced in the first section of this chapter, the AG consists of facts with only *ground terms*, i.e., they do not contain variables. This does not exclude functional terms as we will discuss here. We will, however, present the main restriction of RBL that only functional terms that occur in the AG can also occur in query templates.

4.1. Functional Terms

Functional terms can be partially incorporated into BPEM by associating an AG vertex with each functional term that occurs in the program assertions. This is consistent with our previous thinking since a constant can be considered a functional term of zero arity. Since functional terms can have any arity and be recursive, we introduce *function edges* to connect a functional term with its arguments. These edges are labeled with the function's name and directed towards the argument.

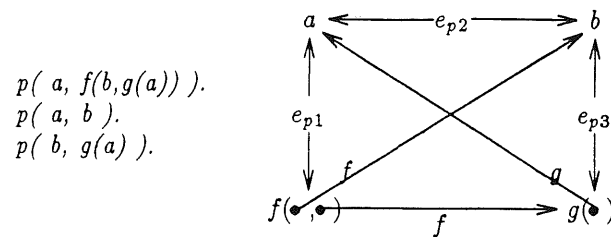


Figure 14

Simple Assertion Graph with Functional Terms

This is illustrated in Figure 14. Three facts appear on the left which for simplicity have the same name p . (In this section, we follow the convention of representing variables with upper-case identifiers and representing constants with lower-case identifiers.) Four different functional terms appear in these three facts: a , b , $g(a)$, and $f(b, g(a))$, having arities 0, 0, 1, and 2, respectively. Even though these three facts are already binary, for completeness, we show them as binarized with the new constant symbols produced by binarization, e_{p1} , e_{p2} and e_{p3} , with edges that identify their arguments. There are three function edges; two for f and one for g . One may note at this point how a predicate argument can be *distributed* over many vertices.

4.2. Functional Terms in the Query

A similar graphical transformation is applied to any queries and clause bodies. Consider Figure 15. A simple query is shown in Figure 15a. Its graphical interpretation is shown in Figure 15b. To establish an order for matching the query to the AG and to identify independent sub-queries, a depth-first spanning tree is found, again ignoring the edge directions since they do not affect the direction of matching. Figure 15c shows one such tree. The spanning tree partitions the graph edges into tree-edges (solid) and back-edges (dashed) and, of course, is rooted, here at a .

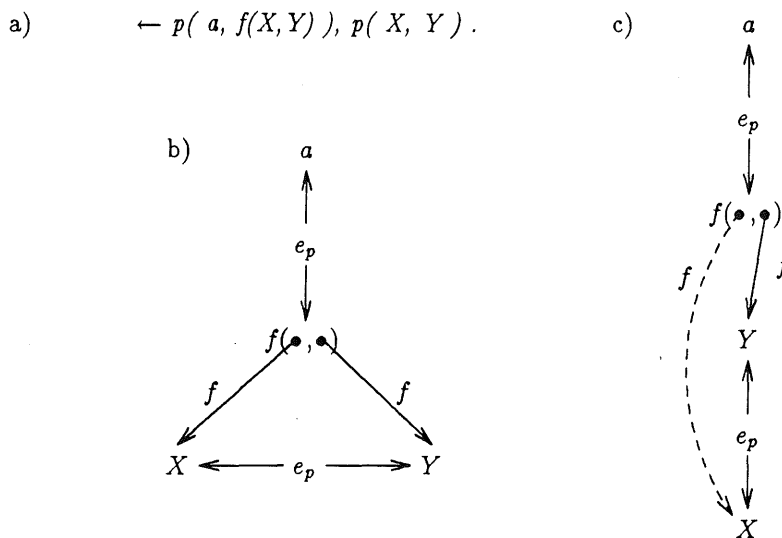


Figure 15

Simple Query with Functional Terms

4.3. Matching without Clause Invocation

The matching process between a QT and the AG proceeds basically the same as before. The QT is injected at an AG vertex matching the QT root and matching continues along the spanning tree edges which could now also be function edges. In Figures 14 and 15c, for example, the initial query is injected at a and the first e_p tree-edge is matched with e_{p1} and e_{p2} . The match at b fails but the PE for $f(\bullet, \bullet)$ matches and sends a message down the f edge to match Y with $g(\bullet)$. In turn, X is matched to b and the PE for b checks that it has a back-edge to $f(\bullet, \bullet)$. Finally, success and the bindings for X and Y are reported back to a .

Several related observations can be made at this point. First, a functional term that contains occurrences of n other functional terms can be distributed over $n + 1$ PEs. For example, the term $f(b, g(a))$ is actually represented by all vertices in Figure 14 and the structure is recorded by the function edges. This means that the unification process is also necessarily distributed over several PEs and may not be completed until other work is done. For example, the unification

for the term $f(X, Y)$ is not complete until the following literal $p(X, Y)$ has been resolved. Hence, each message corresponds to one “call” in the recursive unification algorithm. Only when functional terms are absent, does each message correspond to one step of resolution.

4.4. Clause Head Unification with Functional Terms

An important part of logic programming is the use of unification with functional terms to build new functional terms that don't explicitly appear anywhere in the program, e.g., lists and trees. This is typically implemented by representing the static structure of a program as a *skeleton* and building a stack of *environment frames* where each frame, one per unification, defines the value of each variable occurring in the clause for that invocation. Each variable value is represented by a *molecule* in the frame which is a skeleton/environment pointer pair. In the course of computation, a new functional term is represented by a forest of pointers going up and down in the stack [BRUY82].

BPEM's distributed nature, however, makes the environment stack model of sequential implementations inappropriate. We have already noted that BPEM must consider unification with facts as a different operation from unification with clause heads. Unifying a functional term in a QT with a fact is easy since it means simply matching each function and predicate edge in the term with the AG according to the QTs spanning tree. Unifying a literal and a clause head where functional terms appear is more difficult for at least three reasons:

- 1) Due to the first property previously discussed to solve the data dependencies, each calling literal must be handled by one PE. This means that the unification with the clause head must be done by that PE and cannot have the distributed nature of unifying with the AG.
- 2) Each PE handles a set of vertices in the AG which corresponds to a set of functional terms which occur in the assertions. If arbitrary functional

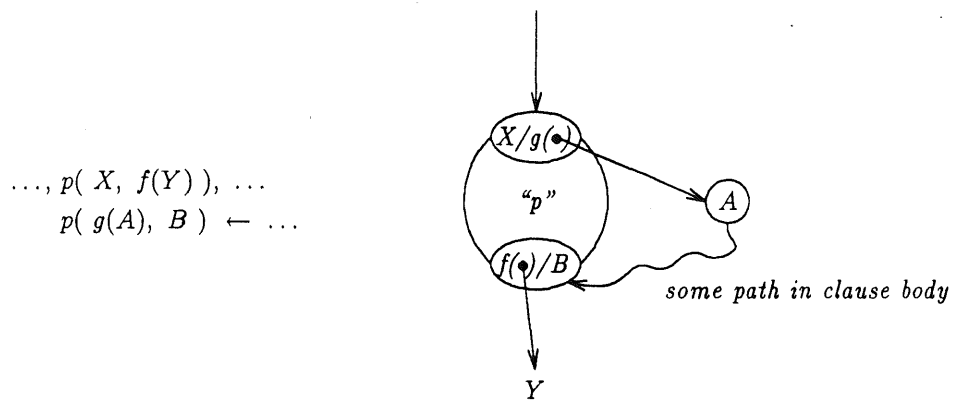


Figure 16

 Head Unification with Functions from the AG

terms could occur in a clause head or a calling literal then completely new functional terms could be built that do not occur anywhere in the AG. For further distributed processing to occur, other PEs needing this new functional term would have to remotely request its value from the creating PE.

- 3) While the *search rule* of BPEM is essentially breadth-first, the spanning tree concept produces a very unorthodox *computation rule*. A spanning tree defines the order in which variables are bound yet there is no guarantee that all variables from one literal are bound before processing another literal. One literal may be partially solved and a second literal worked on before returning to the first. If this was allowed for a clause head, work could begin on parts of the clause body before the head was completely unified, i.e., before it was known that the unification succeeds and that the clause body should be executed at all.

Hence, to maintain the efficiency of BPEM's distributed nature, we will require that the only those functional terms that occur in the AG may occur in query templates or clause bodies. Figure 16 gives an example. where "p" is a macro vertex with two arguments on its boundary that have been unified with the

head of a clause. For X to unify with $g(\bullet)$, the PE handling this unification must have $g(\bullet)$ with a g -edge to some other AG vertex that will bind with A . The body of the invoked clause must then define some binding path from A to B . When B in the clause body is bound, it must be an $f(\bullet)$ vertex with an f -edge to some other vertex that will bind Y . Note that this is really just matching functional terms that already exist in the AG. The functional terms occurring in the calling literal or the clause head can be thought of as just an extension of the goal list or clause body that must also be matched.

4.5. Variables in the Assertion Graph

Variables in the AG are problematic, not because they don't have an interpretation in BPEM but rather because their usefulness in semantic networks and knowledge bases is unclear. Consider the following assertion and query:

$$\begin{aligned} &v(a, X). \\ \leftarrow &v(a, Y), p(Y, Z). \end{aligned}$$

Assume this query was injected at a and has propagated to X . At this point, X and Y are bound. To continue the query, however, Y should be bound to the first term of every p fact. This suggests some type of *broadcast* mode of communication since any other vertex in the AG could bind with Y . This will not be pursued further here since this is considered an atypical operation for semantic networks and knowledge bases and since BPEM is targeted for asynchronous, message-passing systems over which large amounts of base data are distributed.

5. Non-Logical Extensions

The model as described so far is firmly based in logic and is sufficient for a great many applications. To make BPEM more useful in a practical sense, however, we now introduce several non-logical extensions. These include *variable attributes* for semantic networks and syntactic constructs to allow the user to

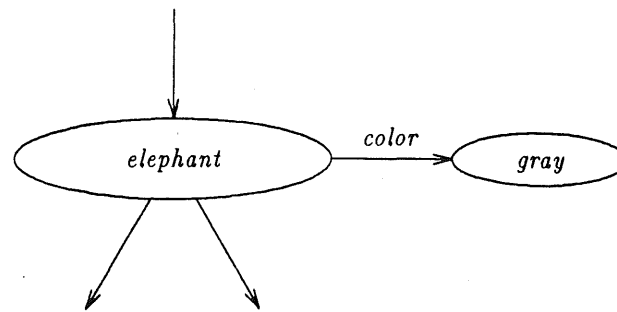


Figure 17

An Attribute

control parallelism by sequentializing AND/OR-branches and to improve efficiency by avoiding unnecessary computation with *cut*.

5.1. Variable Attributes

Attributes are a well-known concept in semantic networks. They are simply some property with some value that is owned by an object. For example,

$$\text{color}(\text{elephant}, \text{gray})$$

means that elephants have the color gray. In RBL, this can be interpreted as simply a logical edge between *elephant* and *gray* as shown in Figure 17. The query

$$\text{color}(\text{elephant}, X)$$

will succeed by binding the variable *X* to *gray*. Over the course of a computation, the value of an attribute will not necessarily remain constant. To change the value of an attribute, it is necessary to (conceptually) *retract* the original clause, e.g., $\text{color}(\text{elephant}, \text{gray})$, and *assert* a new clause, e.g., $\text{color}(\text{elephant}, \text{pink})$. To do this we simply introduce the non-logical predicates *retract* and *assert*:

$$\text{retract}(\text{color}, \text{gray})$$

$$\text{assert}(\text{color}, \text{pink})$$

Since changing the value of an attribute is a common operation, we introduce a special notation for it: If the second term of an attribute literal appears in quotes, it is interpreted as *assigning* that value to the attribute. For example, the literal

color(elephant, "pink")

assigns the value *pink* to the color of elephants regardless of any previous value.

The predicates *assert* and *retract* fill the same niche as in conventional logic languages. We note that *assert* is a no-op if the attribute already exists with the same value and that *retract* fails if the attribute doesn't exist with the proper value. They are non-logical in that their purpose is to have the side-effect of changing the program (the AG) at run-time. In BPEM, as we shall see, attributes are kept locally with its owner. Thus, AG vertices with attributes have a form of memory that can be read and written. While this allows semantic networks to record data and also the state of the query, it also allows race conditions to occur if improperly programmed and multiple (sub)queries assign the same attribute.

5.2. Evaluable Predicates

The value of a variable in the local environment of an AG vertex can be tested with six relational operators written as evaluable predicates. This means that the binding of any variable passed to a vertex in a query or reply message or the value of a local attribute can be compared with another or tested against a constant. These are written as prefix operators:

$$\begin{array}{lll} ==(X, pink) & <(X, \mathcal{I}) & >(X, \mathcal{I}) \\ !=(X, pink) & <=(X, \mathcal{I}) & >=(X, \mathcal{I}) \end{array}$$

These predicates will succeed or fail depending on the current bindings of the variables involved. While these predicates are consistent with the graphical interpretation of BPEM, they are most efficiently evaluated in a non-logical, procedural fashion. For example, if an attribute *X* took an integer value, then testing for inequality with a constant in a logical fashion would require a prohibitive number of edges or require a prohibitive number of inferences to establish the inequality.

5.3. User Control of Parallelism

The other non-logical extensions involve the control of the computation, i.e., the control of parallelism. In BPEM, clauses are written using a nested sub-query syntax where literals at the same level indicate AND-parallelism. As usual, OR-parallelism occurs between clauses or assertions having the same predicate. The user is allowed basic control of parallelism with the following constructs:

- Parallel AND: $\dots p(\bar{x})|q(\bar{y})\dots$
- Sequential AND: $\dots p(\bar{x}), q(\bar{y})\dots$
- Parallel OR: $\dots p(\bar{x})\dots$
- Sequential OR: $\dots p\langle 1, \dots, n \rangle(\bar{x})\dots$
 $\dots p\langle \rangle(\bar{x})\dots$
- Cut: $p(\bar{x}) \leftarrow q(\bar{x}), !, r(\bar{x})$.

A query or clause body is written as a sequence of literals. Every literal that is followed by a vertical bar “|” executes in parallel with the literals that follow. Every literal that is followed by a comma “,” must complete its execution before the next literal begins. This simple scheme is not as powerful as other well-known parallel constructs such as *fork-join* or *cobegin-coend* in that some combinations of sequential-parallel execution are not possible. In the vast majority of situations, however, it should be completely adequate. The simplicity of this syntax is extremely attractive for readability since a recursive syntax (nesting) is already being used for sub-queries.

To control OR-parallelism we introduce an annotation that simply specifies the sequential order in which literals are to be done. In general for any literal $p(a, b)$, all matching literals are done in parallel. For $p\langle 1, \dots, n \rangle(a, b)$, however, literals $1, \dots, n$ based on order of occurrence in the program, are done sequentially. Note that literals could be done in any order and not necessarily from 1 to n . As

a simplification, just $p\langle\rangle(a, b)$ is used to indicate sequential execution from 1 to n as opposed to complete OR-parallel execution.

This allows us to introduce the last non-logical extension: *cut*. If cut “!” occurs in any clause, then all subsequent OR-branches from the previous literal back to the clause head are discarded. This essentially prunes those OR-branches from the search space thus making it smaller. Of course, the PE that actually does the cutting is the one that invoked the clause in which the cut occurred. This means that when a PE encounters a cut, it must return a “cut encountered” status to the (not necessarily proper) ancestor such that the ancestor can prune any remaining OR-branches.

CHAPTER 4

Example Applications

1. Introduction

This chapter gives two larger examples to illustrate BPEMs usefulness. The first is a geographical database. This will illustrate the basic aspects of building a database, i.e., the AG, and queries and clauses. More importantly, this will illustrate the fact that an extensional database is a set of relations between domain sets that could quite easily grow very large and, hence, would benefit from a massively-parallel system. The second is a real-time monitoring system which illustrates the use of BPEM for semantic networks. This example is longer and more involved since it uses many of the non-logical extensions to capture the behavior of a real-time system. Each example will be followed by a discussion.

2. A Geographical Database

2.1. The Assertion Graph

A relatively small database of world geography was built to test cost estimates for BPEM queries. This database has the domains and predicates listed in Table 1 with the indicated sizes. It is graphically depicted in Figure 18 where each line represents a set of tuples. While this database is certainly not geographically exhaustive it is large enough to support queries generating thousands of query messages.

The predicate names are self-explanatory. Assertions such as `borders(france,mediterranean)` and `isacityin(naples,italy)` appear in the

size	Domain Sets (7 sets with 151 elements)
6	Continents
4	Oceans
22	Seas
42	Countries
56	Cities
16	Rivers
5	Set_Names
size	Predicates (13 predicates with 860 tuples)
260	borders(Countries \cup Oceans \cup Seas, Countries \cup Oceans \cup Seas)
104	isa(Continents \cup Oceans \cup Seas \cup Rivers \cup Cities, Set_Names)
42	isacountryin(Countries, Continents)
56	isacityin(Cities, Countries)
42	iscapitalof(Cities, Countries)
40	flowsby(Rivers, Countries \cup Cities)
16	flowsinto(Rivers, Oceans \cup Seas \cup Rivers)
104	ais(Set_Names, Continents \cup Oceans \cup Seas \cup Rivers \cup Cities)
42	hascountry(Continents, Countries)
56	hascity(Countries, Cities)
42	hascapital(Countries, Cities)
40	hasriver(Countries \cup Cities, Rivers)
16	drains(Oceans \cup Seas \cup Rivers, Rivers)

Table 1

World Geography Database

database. There is a special domain set, called 'Set_Names', that is used with the isa and ais predicates. This includes facts such as isa(atlantic,ocean) and its inverse ais(ocean,atlantic). This allows us to query an entire domain set. The reader may note that borders is the only symmetric predicate. The other predicates are paired for traversing a relation in either direction.

2.2. Queries and Clauses

Three possible queries that could be put to this database have the English equivalents:

- (1) "Which cities are in countries on an ocean?"
- (2) "Which triplets of countries border on each other?"

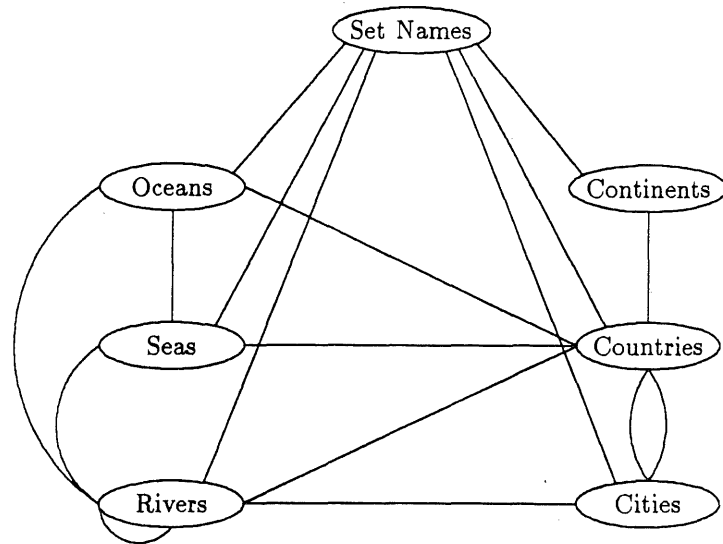


Figure 18

World Geography Database

- (3) "Which cities are on a river that flows into an ocean after flowing through exactly one other country downstream?"

These could be written in logic as:

```
?- ais(city, City), isacityin(City, Country),
   borders(Country, Ocean), isa(Ocean, ocean).
```

```
?- ais(country, Country1), borders(Country1, Country2),
   borders(Country2, Country3), borders(Country3, Country1),
   isa(Country2, country), isa(Country3, country).
```

```
?- ais(city, City), hasriver(City, Riv), flowsinto(Riv, Ocn),
   isa(Ocn, ocean), borders(Ocn, Coun1), hasriver(Coun1, Riv),
   isacountryin(Coun1, Cont), borders(Coun1, Coun2),
   hascity(Coun2, City).
```

respectively, and have the graphical RBL representation as shown in Figure 19.

Three possible clauses that could be defined for this database have the English equivalents:

- (1) "Does this city have ocean access via a river?"
- (2) "Is this city a capital on a river?"

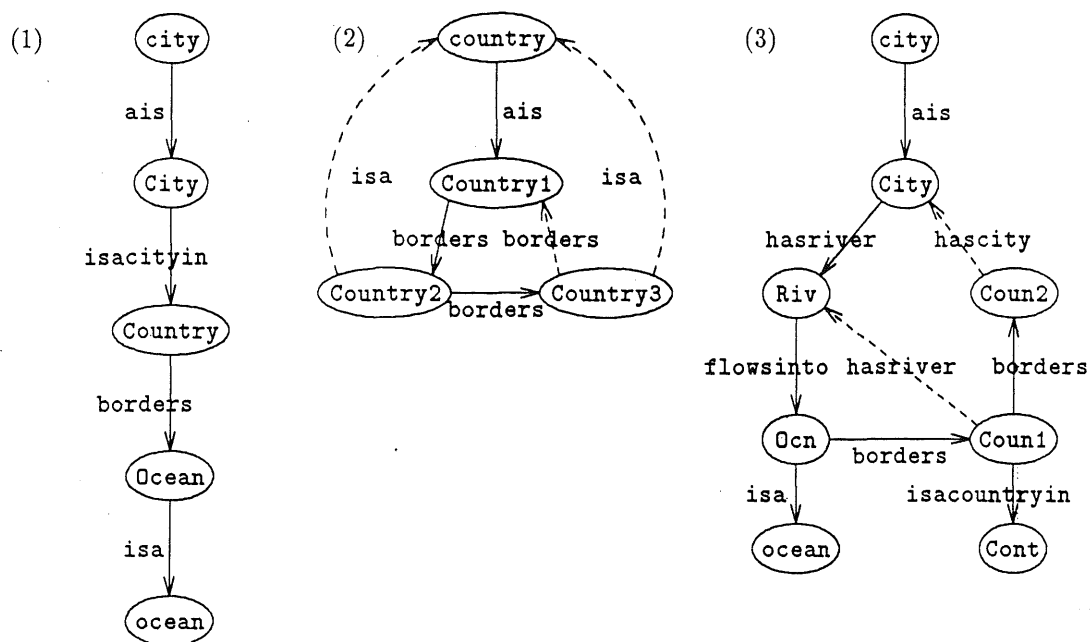


Figure 19

Three Queries in RBL

(3) "Which cities are on the same river in bordering countries?"

These could be written in logic as:

```
ocean_access(City,Ocean) :- hasriver(City,River),
    flowsinto(River,Ocean), isa(Ocean,ocean).
```

```
capital_on_river(City,River) :-
    iscapitalof(City,Country), hasriver(City,River).
```

```
rivercities(City1,City2) :- hasriver(City1,River),
    flowsby(River,City2), isacityin(City2,Country2),
    borders(Country2,Country1), hascity(Country1,City1).
```

respectively, and have the graphical RBL representations as shown in Figure 20.

2.3. Discussion

The construction and use of this database is relatively straight-forward. It could be made extremely large simply by making the geographical data more complete and extensive. Queries, such as (1), (2) and (3), can be answered by

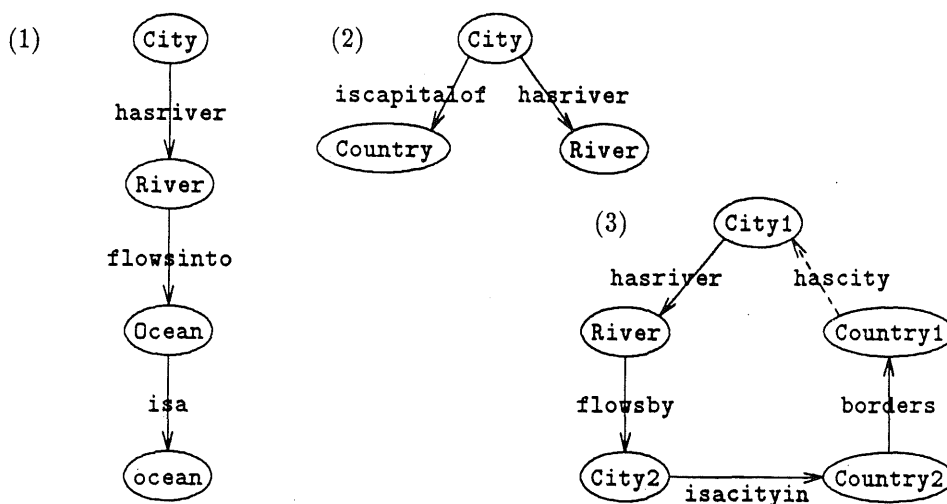


Figure 20

Three Clauses in RBL

injecting them into the AG vertex matching their root vertices and matching the spanning tree edges (solid arrows) until the leaves are reached and all back-edges (dashed arrows) have been checked. Clauses, such as (1), (2) and (3), can be used to construct larger queries.

The efficiency of this database, however, could be improved. Since some relations, such as `borders`, connect several domain sets, it is sometimes necessary to check whether a variable is bound to the desired type of constants. In query (1), for example, it is necessary to check whether `Ocean` is actually bound to an ocean name rather than to a country or sea name. A similar situation occurs for `Country1` and `Country2` in query (2). This is an example of where a many-sorted logic would improve efficiency by restricting the types of constants that can bind to a given variables. This can also be avoided in a simpler way by using specific relations to relate just two domains. One relation can be used for oceans bordering oceans and another for oceans bordering countries, etc. The broader interpretation, however, can be retained by a new predicate, say `borders_all`, that invokes a set of clauses which name the specific relations in their bodies.

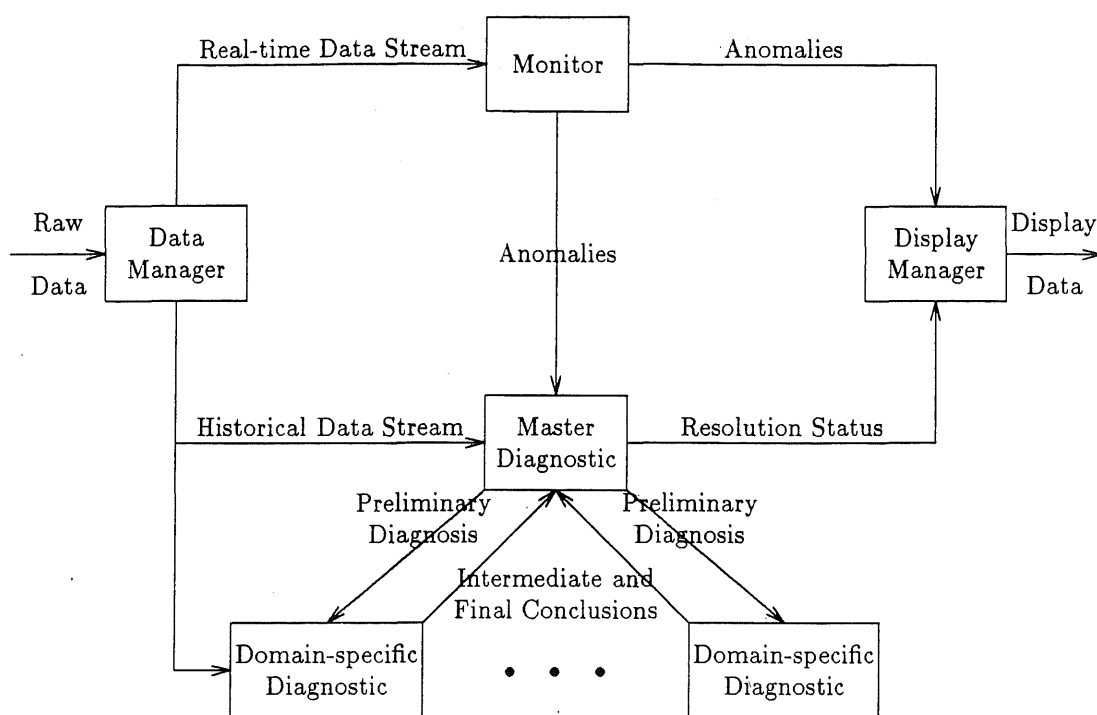


Figure 21

The Generic KNOMES Shell

3. RMS_KNOMES

The second example involves a remote manipulator system that is currently being developed by the McDonnell Douglas Astronautics Corporation for the NASA space station. This system will have an on-board monitoring system known as the Remote Manipulator System Knowledge-Based Maintenance Expert System, RMS_KNOMES for short. RMS_KNOMES is a non-classified research project and thus we were able to obtain the specifications as well as some actual code for the knowledge base from McDonnell Douglas [MDAC87]. This is an excellent example application for BPEM that demonstrates virtually all aspects of the model; especially the use of the non-logical extensions to handle continuous streams of real-time input data.

3.1. Overview

The overall idea of RMS_KNOMES is shown in Figure 21[FRANK87]. The real-time data stream into the Monitor consists of message tuples with the form $\langle message_id, time, content \rangle$. The field *message_id* identifies the message type (not the individual message). The field *time* is simply a time-stamp and *content* is simply some data meaningful to the given type. The function of the Monitor is to detect anomalies in this data stream. When this happens, it passes the anomaly to the Master Diagnostic which identifies the sub-systems that could be responsible. The anomaly is then passed to the Domain-Specific Diagnostic for detailed analysis. This whole process is chronicled by the Display Manager. The two modules of interest to us are the *Monitor* and the *Master Diagnostic*.

3.1.1. The Monitor – Contexts, Scripts and Expectations

To detect anomalies, the Monitor uses the idea of *contexts* and *scripts* to form *expectations* of what it should see in the data stream. There is a hierarchy of contexts that describe general modes of operation for the manipulator. Each context can have any number of scripts associated with it that describe the expected sequence of events for various operations in that context.

Contexts are enabled or disabled by specific enable/disable messages on the input stream. (Note that more than one can be enabled simultaneously.) When a message occurs that signals the initial event in a script, that script is enabled *if* its context is enabled. (Note that a script may require more than one context to be enabled.)

In addition to the script-based anomalies, built-in test equipment can generate *simple anomaly messages* on the data stream that explicitly denote anomalies; the Monitor recognizes these anomalies by simply recognizing their message types. Hence, the anomaly types that can be detected are summarized as follows:

- 1) Script-based event timed-out. The expected event did not occur within the proper time window.
- 2) Unexpected script-based events. An event occurred without the properly enabled scripts.
- 3) Simple anomalies.

Note that the first anomaly is categorized as an *expected change* while the second and third are both categorized as an *unexpected change*.

In conclusion, the message types that drive the Monitor are summarized as follows:

- 1) Enable/disable contexts,
- 2) Event messages, and
- 3) Simple anomaly messages.

3.1.2. *The Master Diagnostic – Command Chain Models*

When the Monitor detects an anomaly, it sends that information to the Master Diagnostic (MD). The job of the MD is to identify the possible sources of the anomaly. This is done by using a hierarchy of *command chain models*. A command chain model is a linear chain that shows the sequence of commands that should occur within a given *context*; the same contexts as for scripts. These models can be very general or very specific and can be placed into a hierarchy by relating the vertices of the more general and more specific models. Hence, a sequence of commands would follow the horizontal edges while models of varying generality would be related by the vertical edges. Note that a relation and not a mapping exists between models since the vertices may not fall into a vertical one-to-one correspondence. Also, a command chain model is not required to be complete; it may have missing horizontal edges.

When the MD receives an anomaly message, it conceptually enables the model corresponding to the context in which the anomaly occurred *and* all of the more

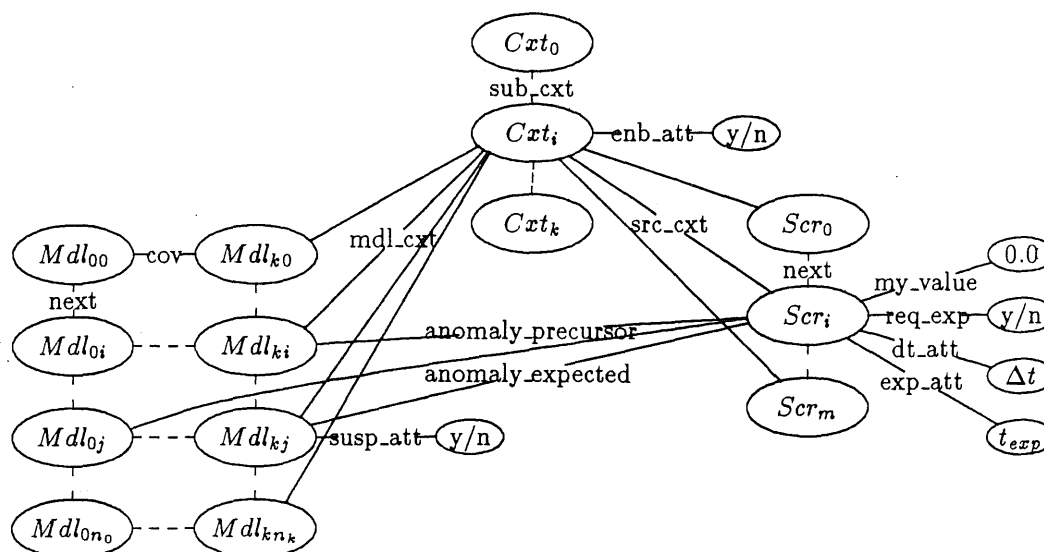


Figure 22

Form of Semantic Network for RMS_KNOMES

generic contexts. If the anomaly is an expected change, then the precursor event and the expected event have a predefined mapping to left and right *edges* in each model, respectively. All vertices between each pair of left-right edges then become *suspect* sources of the anomaly. If the anomaly is an unexpected change, then the unexpected event is mapped into each model and all “upstream” vertices to the left become suspect. Since each model may be incomplete, following the vertical links between models from suspect vertices, in addition to following the horizontal links, may find other suspect vertices.

Each suspect vertex corresponds to a specific hardware entity that becomes suspect. (Note that if the anomaly is simple, an entity is directly made suspect.) This preliminary diagnosis is then given to a Domain-Specific Diagnostic for a more detailed examination.

3.2. A Semantic Network for RMS_KNOMES

Contexts, scripts and models all have a graphical representation which makes BPEM a potential vehicle. A general description of a semantic network for

```

enb_cxt_msg( X ) :-
    sub_cxt( Y, X )[ enb_att( Y, yes ) ],
    enb_att( X, "yes" ).

disab_cxt_msg( X ) :-
    enb_att( X, "no" ),
    scr_cxt( X, S )[ exp_att( S, "0.0" ) ] |
    sub_cxt( X, Y )[ disab_cxt_msg( Y ) ].

```

Figure 23

Context Clauses Initiated at *Cxt* Vertices

specific context. Event messages signal the occurrence of a specific script event. Simple anomaly messages directly identify suspect vertices in the chain models and don't require further processing by the Monitor. In the following discussion, we present the clauses for processing context and event messages and the clauses for handling anomalies that have been detected by script vertices. The message type and content determine which context or script vertex is its destination. This gives rise to two possibilities for injecting queries: (1) a dedicated PE listens to the data stream and knows how to directly route each message to its final destination, or (2) data stream messages are broadcast to all PEs which listen for their own. Which method is used is unimportant here. Copies of the context and event clauses are located at each context and script vertex, respectively. A copy of the model clauses for handling anomalies is located at each model vertex.

Context Messages. Figure 23 shows the clauses for enabling/disabling a context. The `enb_cxt_msg` and `disab_cxt_msg` clauses are invoked, respectively, whenever an enable/disable message has been recognized by a context node. When enabling a context, the PE first checks that the parent context of `X` is enabled and only then sets `X`'s enable attribute (`enb_att`) to "yes". Similarly, to disable context, the PE handling (`X`) sets its enable attribute to "no", removes all expectations of dependent script vertices by setting the expectation attribute `exp_att` to "0.0", and then recursively sends the disable message to all descendant contexts. Attribute

RMS_KNOMES is given in Figure 22 with contexts, scripts and models in large ellipses and their associated attributes in small ellipses. The context vertices, labeled Cxt_i , can be in a tree structure and each context has an attribute that records whether the context is enabled or not. Each context can enable many script vertices, labeled Scr_i , each of which has four attributes. (These are explained in the next section.) The chain models are shown here from top to bottom with more general on the left and more specific on the right with vertices labeled Mdl_k , each of which have a suspect attribute. The model vertices are enabled by the same contexts and each script vertex can detect an anomaly that has a precursor event and an expected event in the command chain models. As noted previously, model vertices that lay between these two boundaries correspond to the suspected hardware modules. As defined in the previous chapter, each context, script and model vertex is conceptually an active processing element (PE) capable of communicating with its neighbors.

3.3. BPEM Logic Clauses for RMS_KNOMES

For the purpose of developing these clauses, we assume that each message on the real-time data stream initiates the processing of a clause as if each real-time message was a literal in a query that matches the clause head. Since invoking literals are represented as one vertex in a query template, they do not need to be binary or binarized. Hence, the primary purpose of each data stream message is to provide an initial binding environment for the clause body via their unification. In addition, since messages are received one at a time, this means that entire queries are *incrementally received in a non-deterministic order*. Thus, attribute variables will essentially be used to record *state* information about the messages that have been seen so far.

As mentioned previously, there are three types of real-time messages: *context*, *event* and *simple anomaly* messages. Context messages either enable or disable a

supplies the initial bindings of `Scr_Id`, `Val` and `Time`. According to the clause body, the script vertex first checks for the proper content value and gets the value of its context's enable attribute. The clause `test_enb` is called and two clauses are invoked for the cases `yes` and `no`. If the context is not enabled, then an unexpected anomaly message is sent to the associated model vertex. This message corresponds to the literal `unx_anc(Scr_Id, Mdl)`. If it is enabled, then the expectation required attribute (`req_exp_att`) is tested in the same OR-branch manner. If no expectation of this event is required, then the allowed time until the next event in this script (the Δt value of the `dt_att` attribute) is added to the current `Time` and used to set the expectation attribute (`exp_att`) of all subsequent vertices in this script. If an expectation is required, then the value of the expectation attribute is tested. If there was no expectation (`Eta == 0.0`) then another unexpected anomaly message is sent. If `Eta < Time`, then the event did not occur within the proper time window and an expected anomaly message is sent. If the event did occur within time (`Time <= Eta`), then the expectation is removed and the expectation attributes of all subsequent script vertices are set as before.

Anomalies. The event clauses can detect expected and unexpected anomalies. Whenever this occurs, the detecting script vertex initiates the `exp_anc` or `unx_anc` clause, respectively, in the model vertex that is specifically associated with the detected anomaly. For every expected event anomaly, there is a precursor event and an expected event. These two events give two boundaries in the chain models between which all vertices represent sub-systems suspected of causing the anomaly. In this case, the clauses in Figure 25 are used to set the `susp_att` attribute of all suspect vertices. (Of course, the suspect attribute could be a vector of Booleans if many anomalies could occur simultaneously.) In `exp_anc`, the model's context is first checked to make sure it is enabled. Then the `susp_att` is set if it is not set already. Next, `check_boundary_1` is called *sequentially* in order to make use of

```

event_msg( Scr_Id, Val, Time ) :-
    my_value( Scr_Id, Val ),
    src_cxt( Cxt, Scr_Id )[ enb_att( Cxt, Enb ) ],
    test_enb( Scr_Id, Enb, Time ).

test_enb( Scr_Id, Enb, Time ) :-
    ==( Enb, no ),
    anomaly_precursor( Scr_Id, Mdl )[ unx_anc( Scr_Id, Mdl ) ].

test_enb( Scr_Id, Enb, Time ) :-
    ==( Enb, yes ),
    test_req_exp( Scr_Id, Time ).

test_req_exp( Scr_Id, Time ) :-
    req_exp_att( Scr_Id, no ),
    dt_att( Scr_Id, Td ),
    +( Time, Td, Next_Time ),
    next( Scr_Id, Next_Id )[ exp_att( Next_Id, "Next_Time" ) ].

test_req_exp( Scr_Id, Time ) :-
    req_exp_att( Scr_Id, yes ),
    exp_att( Scr_Id, Eta ),
    test_eta( Scr_Id, Eta, Time ).

test_eta( Scr_Id, Eta, Time ) :-
    ==( Eta, 0.0 ),
    anomaly_precursor( Scr_Id, Mdl )[ unx_anc( Mdl, Scr_Id ) ].

test_eta( Scr_Id, Eta, Time ) :-
    <( Eta, Time ),
    anomaly_precursor( Scr_Id, Mdl )[ exp_anc( Mdl, Scr_Id ) ].

test_eta( Scr_Id, Eta, Time ) :-
    <=( Time, Eta ),
    exp_att( Scr_Id, "0.0" ),
    dt_att( Scr_Id, Td ),
    +( Time, Td, Next_Time ),
    next( Scr_Id, Next_Id )[ exp_att( Next_Id, "Next_Time" ) ].

```

Figure 24

Event Clauses Initiated at *Scr* Vertices

operations are always local and done first but the second and third operations can be done in parallel as indicated by the vertical bar.

Event Messages. Figure 24 shows the clauses that handle event messages. An event message initiates the `event_msg` clause in a specific script vertex and

```

unx_anc( A, Id ) :-
    mdl_cxt( A, Cxt )[ enb_att( Cxt, yes ) ],
    susp_att( A, no ),
    susp_att( A, "yes" ),
    covers( Apar, A )[ unx_child( Apar, Id ) ] |
    covers( A, Achild )[ unx_par( Achild, Id ) ] |
    next( A, Anext )[ unx_anc( Anext, Id ) ].

unx_child( A, Id ) :-
    mdl_cxt( A, Cxt )[ enb_att( Cxt, yes ) ],
    susp_att( A, no ),
    susp_att( A, "yes" ),
    covers( Apar, A )[ unx_child( Apar, Id ) ] |
    next( A, Anext )[ unx_anc( Anext, Id ) ].

unx_par( A, Id ) :-
    mdl_cxt( A, Cxt )[ enb_att( Cxt, yes ) ],
    susp_att( A, no ),
    susp_att( A, "yes" ),
    covers( A, Achild )[ unx_par( Achild, Id ) ] |
    next( A, Anext )[ unx_anc( Anext, Id ) ].

```

Figure 26

Mdl Clauses for Unexpected Anomalies

For unexpected anomalies, only the upper boundary exists. In this case, the clauses in Figure 26 are used to make all lower vertices suspect. This means that messages propagate until there are no further edges to check. Note that it makes no difference if these messages are propagated in an OR-parallel or an AND-parallel fashion since it is their *side-effects* on the attribute variables which are important. The success or failure of these clauses is immaterial. This is the case in general for clauses dealing with attribute variables. Reply messages, however, should be sent because an ancestor PE may be executing branches sequentially and would need the reply messages for synchronization, i.e., starting the next branch on receipt of a reply message from the previous branch.

3.4. Discussion

This example application is very different from the first, especially in its use of the non-logical extensions. Rather than inferring statements from a static

```

exp_anc( A, Id ) :-
    mdl_cxt( A, Cxt ) [ enb_att( Cxt, yes ) ],
    susp_att( A, no ),
    susp_att( A, "yes" ),
    check_boundary_1<>( A, Id ).

check_boundary_1( A, Id ) :- anomaly_expected( A, Id ), !.
check_boundary_1( A, Id ) :-
    covers( Apar, A ) [ exp_child( Apar, Id ) ] |
    covers( A, Achild ) [ exp_par( Achild, Id ) ] |
    next( A, Anext ) [ exp_anc( Anext, Id ) ].

exp_child( A, Id ) :-
    mdl_cxt( A, Cxt ) [ enb_att( Cxt, yes ) ],
    susp_att( A, no ),
    susp_att( A, "yes" ),
    check_boundary_2<>( A, Id ).

check_boundary_2( A, Id ) :- anomaly_expected( A, Id ), !.
check_boundary_2( A, Id ) :-
    covers( Apar, A ) [ exp_child( Apar, Id ) ] |
    next( A, Anext ) [ exp_anc( Anext, Id ) ].

exp_par( A, Id ) :-
    mdl_cxt( A, Cxt ) [ enb_att( Cxt, yes ) ],
    susp_att( A, no ),
    susp_att( A, "yes" ),
    check_boundary_3<>( A, Id ).

check_boundary_3( A, Id ) :- anomaly_expected( A, Id ), !.
check_boundary_3( A, Id ) :-
    covers( A, Achild ) [ exp_par( Achild, Id ) ] |
    next( A, Anext ) [ exp_anc( Anext, Id ) ].

```

Figure 25

Mdl Clauses for Expected Anomalies

the *cut*. If an *anomaly_expected* edge exists between this model vertex and the originating script vertex, then the lower bound has been reached and no further messages are sent. If this is not the case, then the next *check_boundary_1* clause is invoked which propagates messages left, right and down in the models. Similar sets of clauses are used when the message comes from the right and is only propagated left and down and when the message comes from the left and is only propagated right and down.

CHAPTER 5

Implementation and Architectural Issues

1. Introduction

Chapter 3 introduced BPEM and RBL by conceptualizing the AG as a network of PEs that communicate via message-passing. This very natural, straightforward interpretation of graph matching makes some clear choices on the general design issues for parallel systems discussed in Chapter 2.

- 1) This will be a *data-driven* system since every message can be processed immediately after arrival which may, in turn, initiate other messages.
- 2) The problem will be *decomposed* along the lines of resolution and unification. Each query message will cause some relatively small number of unifications to be done.
- 3) Since a typical query could consist of hundreds or thousands of unifications or more, the *granularity* will still be rather fine. It is, however, not as fine as it could be. Each unification is potentially part of many different steps of resolution (when initiating parallelism). Furthermore, since the number of unifications per message can vary, so does the grain size.
- 4) *Scheduling* can be done simply by using a FIFO queue; messages can be processed in the arrival order. As long as the AG is considered to be an active network of PEs, there is really no *allocation* of AG vertices to PEs that needs to be done. As soon as we discuss supporting architectures, however, allocation becomes very important since the AG may have a different topology and be much larger.

knowledge base, the side-effects of assigning to variable attributes are used to dynamically change the knowledge base. This is done for enabling and disabling contexts, for setting and resetting the expectation of events, and flagging suspect sub-systems. These attributes are tested using the relational predicates and new expected time values are computed using the arithmetic predicates.

There are several other important observations to be made. The first is that in many cases, reply messages are not needed. Whether a literal or clause succeeds or fails is secondary to the side-effect of setting an attribute. Secondly, this makes "incremental" queries possible. Literals that have a logical effect on one another can be processed independently when they become available. A further effect is that the distinction between forward and backward chaining is blurred. While each clause is backward chaining, the overall computation is a forward chaining of attribute assignments. A fair question to ask at this point is whether this is really logic programming any more? One answer is that this is as much logic programming as Prolog is since the same non-logical capabilities that diverge from the mathematical basis of resolution theorem-proving, exist in Prolog. Regardless of the usage, resolution still remains as the *inference mechanism*.

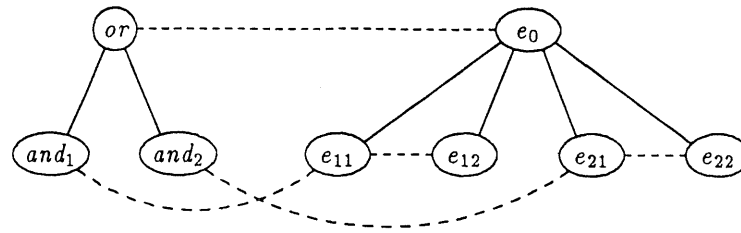


Figure 27

A General Layer of the Activation and Environment Trees

An activation tree is built in a distributed manner over the network as the computation proceeds. Every edge from an AND-node to a descendent OR-node corresponds to a match between the AG and QT or a clause invocation. If an edge corresponds to a match, then the sub-tree below this edge possibly resides on a different processor. (In the section on supporting architectures, we will see why this is possible but not necessary.)

In parallel with the activation tree is the *environment tree*. Every OR-node has a binding environment (BE) that is an array of binding pairs. The contents of this BE are defined by the match or clause invocation that produced the OR-node. Every AND-node, in general, has a list of BEs whose parent is the OR-node environment. While AND-branches share the parent environment, they cannot “see” the environments of their siblings since each represents an independent sub-query with no shared variables.

A list of BEs can result when AND-branches are done sequentially. Consider Figure 27 and assume that the predicates p_1, p_2 have to be done sequentially from branch and_1 . Predicate p_1 may return more than one solution which would be recorded in environments e_{11} and e_{12} . Predicate p_2 then has to be solved twice; once for the environment $e_0 \circ e_{11}$ and once for the environment $e_0 \circ e_{12}$. We note that if AND-sequential execution is not used, then the activation and environment tree are isomorphic. In this case, storage of the binding information could be simplified by keeping it with vertices of the activation tree.

- 5) A loosely-coupled, message-passing method of *communication* is central to achieving a highly-parallel, distributed system that is not constrained by a tightly-coupled, shared-memory design. We also note that BPEM does not usually require any *synchronization*. This is only necessary when clause invocation causes conflicting binding orders that must be resolved.
- 6) Finally, a variety of *network topologies* are feasible assuming that the allocation can be done properly.

This chapter will develop these design choices with the dual purpose of (1) investigating their ramifications and (2) making the model concrete enough to simulate. We will begin by identifying how the computation will be structured by using message-passing and concomitantly what information a message must carry. We will then define basic procedures for using these messages, i.e., the “code” that each PE will run. Supporting network architectures are discussed next including the topics of allocation and routing. Miscellaneous topics are discussed last.

2. Computation via Message-Passing

2.1. The Activation and Environment Trees

We have introduced BPEM as a process of matching a tree with back-edges to a graph. Since the solutions produced may be completely distinct or share common sub-solutions, a tree is the natural data structure for recording this process. This will be called the *activation tree*. This is essentially an AND/OR-tree that corresponds to the search space of the query. Every vertex in a query template and every clause invocation is associated with an OR-node and the immediately descendent level of AND-nodes. Each OR-node and its sub-tree is associated with one specific alternative solution. Similarly, each AND-node is associated with an independent partial solution under its parent OR-node. The root of an activation tree is an OR-node.

and Figure 28b shows the query $\leftarrow p_1(a, X)[p_2(X, Y) \mid p_3(X, Z)]$. The QT and the activation tree, Figure 28c, are both rooted at a . Since there is only one tree-edge from a , there is only one AND-node and a conceptually empty environment. There are two different matches for p_1 , however, so there are two OR-nodes which indicate two different bindings for X . From X , there is AND-parallelism between p_2 and p_3 and also OR-parallelism within each one. This produces the remainder of the trees. A solution can be found by following any successful OR-branch and following all successful AND-branches. The four possible solutions here are:

$$\begin{array}{ll} \{X/b, Y/d, Z/k\} & \{X/c, Y/f, Z/k\} \\ \{X/b, Y/e, Z/k\} & \{X/c, Y/g, Z/k\}. \end{array}$$

2.2. Static and Dynamic Data

For efficiency, we must try to minimize the amount of data that must be communicated between parts of the network in order to build the activation and environment trees. That is to say, we have to determine which data must travel between PEs and which can be kept in one place. An important observation is that static data (data that does not change during a computation) should be initialized once while dynamic data must, by its nature, be communicated during the computation.

The static data here are the AG and the QTs which includes the initial query. This constitutes the *program* and it can be partitioned or replicated over the PEs only once at the beginning of the computation. The dynamic data here are the *activations*, specifying which vertices of the QTs are being processed, and the *binding environment*. This constitutes the execution of the program and must be communicated as it is generated.

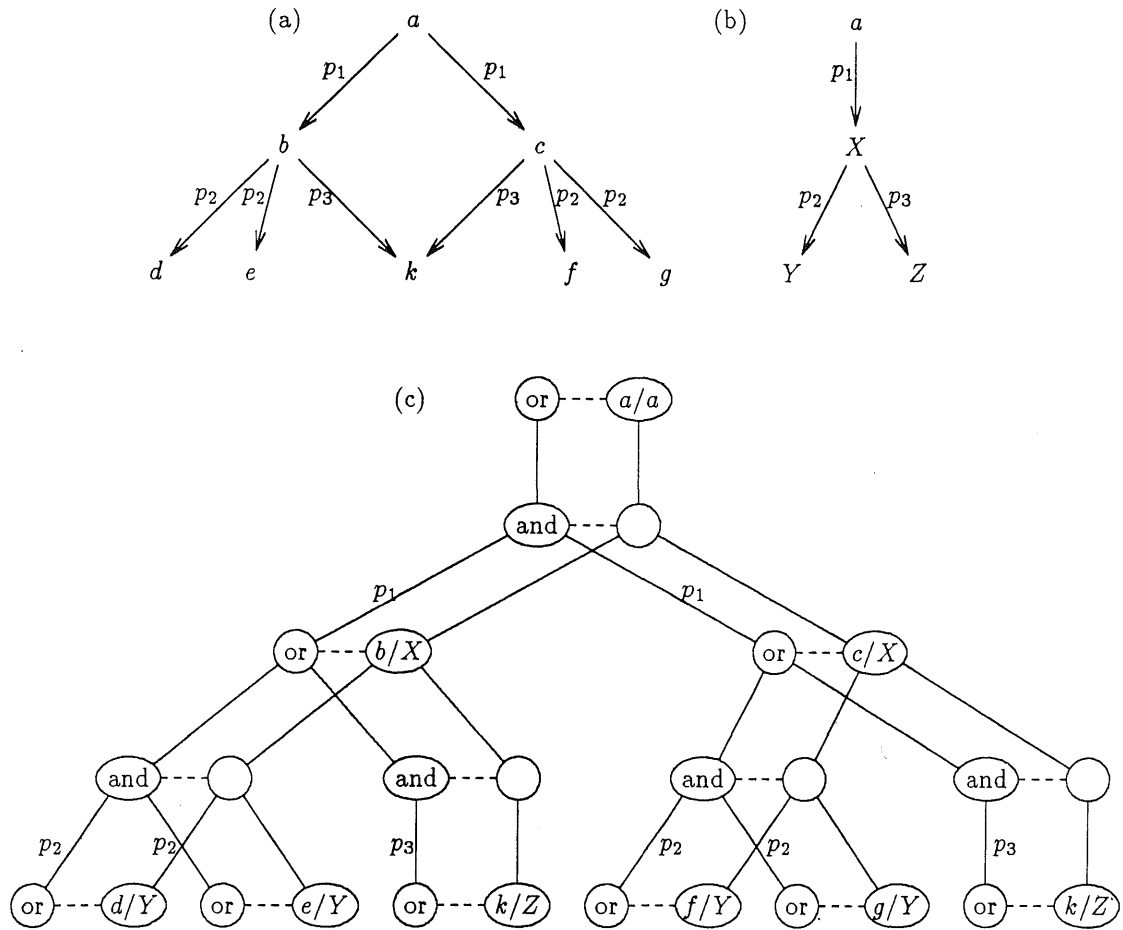


Figure 28

An Example of the Activation and Environment Trees

We now give an example of the activation and environment trees that demonstrates AND-parallelism and OR-parallelism. Figure 28a shows the AG corresponding to the facts

- | | | |
|--------------|--------------|--------------|
| $p_1(a, b).$ | $p_2(b, d).$ | $p_3(b, k).$ |
| $p_1(a, c).$ | $p_2(b, e).$ | $p_3(c, k).$ |
| | $p_2(c, f).$ | |
| | $p_2(c, g).$ | |

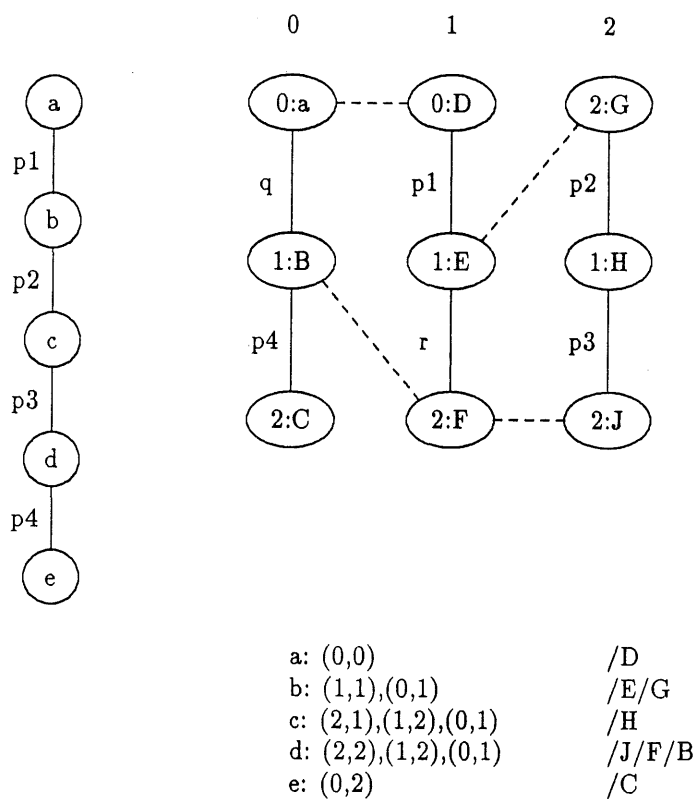


Figure 29

Activation Stack Example

$$\leftarrow q(a, B)[p4(B, C)].$$

$$q(D, F) \leftarrow p1(D, E)[r(E, F)].$$

$$r(G, J) \leftarrow p2(G, H)[p3(H, J)].$$

are shown at the right in tree form and are identified by a query index at the top of the figure and a vertex index within each ellipse. These two indices together constitute a QT tuple. The dashed lines indicate the unifications that are made between clauses during the invocations. The activation stacks of tuples that are sent to each AG vertex are shown at the bottom of the figure along with the bindings that are made in the course of execution. The initial message sent to a is a stack of one tuple that tells it to process clause body 0, vertex 0. To proceed, a must invoke q . So it binds to D and matches the first edge of clause 1. Hence, b is

2.2.1. Partitioning/Replicating the Program

One of the explicit assumptions of BPEM is that the base data in a program, i.e., the AG, is so large that replicating it among all PEs is impractical. Hence, we will always partition the AG over a network of PEs. The QTs, however, are assumed to be relatively small and few. Here, replication of the QTs over all PEs, or rather replication of the QTs to some memory local to each PE, is feasible. Since each AG vertex can participate in only a subset of all literals, only those QTs in which those literals occur would have to be replicated for the PE handling that AG vertex. Since we partition AG vertices over PEs, however, each PE would probably receive a copy of most QTs. Hence, the space saved by doing this would probably be small so we will always replicate all QTs over all PEs.

Since the structure of the QTs are static, every QT and each vertex within a QT can be identified by an index. Hence, every QT vertex will be represented by a tuple of indices: (i, j) . Such a tuple identifies the j -th vertex of the i -th QT.

2.2.2. Activations

Every query message that starts an activation in an AG vertex will carry a QT tuple (i, j) that identifies which QT vertex it is to match. This is simple and straight-forward. Nested clause invocation, however, requires that processing on those clauses be continued when a term in the clause head is matched. Hence, the query and clause vertices being processed can be specified by a stack of QT tuples that reflects the currently invoked clauses. While the size of this stack is technically unbounded, the stack required for an average semantic network or knowledge base query is not expected to be excessively large. In any case, this is clearly preferable to actually transmitting entire graph templates.

Example: Figure 29 shows an example of an activation stack involving two clause invocations. The AG is the chain on the left side. The initial query and clauses

$$q(D, E) \leftarrow p1(D, E) \mid p2(D, F).$$

in the same format as before. The initial message is sent to a which invokes q and binds to D . The stack element for the calling clause is only sent along edge $p1$, however, and not edge $p2$ since F does not occur in the head of q . While c is binding to F , b is binding to E , popping the stack, binding to B and sending a message to d . Vertex d binds to C and finishes the query.

In the most general case, $n - 1$ head terms could appear on $n - 1$ different branches and each branch could require a different subset of bindings. Hence, when processing at any particular vertex, it would be optimal to “split” the received activation stack vertically and sent only the required subset down each descendent tree-edge. This is difficult, however, because every order of clause invocations requires a different “split”. Rather than compute this at run-time, the current implementation simply passes the entire stack down all branches that contain a head term. While this may increase the average message length, the execution time may remain the same since not splitting the stack compensates for time spent processing the longer messages. In addition, n is not expected to be much larger than two in practice.

2.2.3. Binding Environments and Binding Scope

The *binding environment* is the set of bindings necessary for a PE to process a query message. Not surprisingly, this is only a subset of all prior bindings that have been made. A variable X may have many solutions and, hence, many bindings. Any sub-problem of X , however, must be solved independently for each such binding. Thus, any activation within one of the independent sub-solutions can observe only one binding of X . Also, when AND-parallelism is present, a prior binding may not be needed in all subsequent AND-branches.

This idea is developed by the concept of each variable having a *binding scope*. The boundary of a variable’s scope starts where its bindings are produced and ends

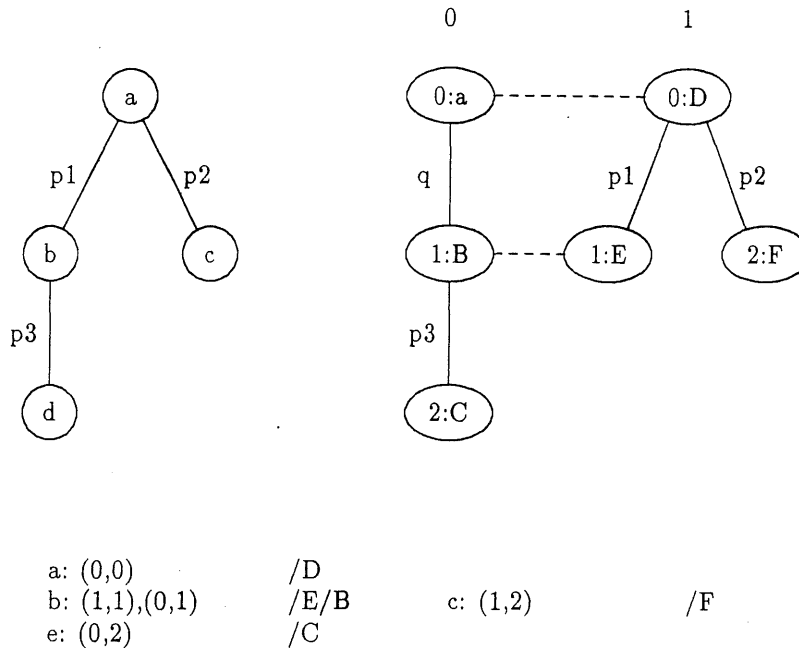


Figure 30

Activation Stack Path Example

sent a message of two tuples telling it to process clause 1, vertex 1, and clause 0, vertex 1. Vertex b binds to E , must invoke r and also binds to G . Similar messages are sent to c and d . When d binds to J , it discovers that clause r is finished, pops a stack element, and binds to F . Clause q is also finished so another stack element is popped and d binds to B . Finally, c is sent a message to process the last vertex in the initial query and it binds to C .

The previous example demonstrates that the processing of a calling clause is resumed when a head term is reached, e.g., J and F . Clauses in RBL are tree-structured, however, and not all branches may contain head terms. Hence, the stack element for the calling clause does not have to be passed down all branches and, in general, different branches will have different stacks.

Example: Figure 30 shows an AG, the initial query and a clause

$$\leftarrow q(a, B)[p3(B, C)].$$

deepest variable to which it is connected by a back-edge. The next two trees show the scopes for B and C . Since D and E have no back-edges to a descendant, their scopes are limited to themselves. The superimposition of these scopes determines the binding environments that must be transmitted between vertices along the tree-edges to check for back-edges as discussed above. The binding environments for this example are shown as sets at the far right. A must be sent as far as D , B and C must be sent to E but D and E need never be sent.

While bindings are clearly dynamic, *scope is static*. They can be determined once at the beginning of the computation. Hence, it is possible to determine at each activation, which bindings need to be transmitted. Consider that the processing of an activation always commences with a specific binding environment. It will always use the binding from its parent. If there are any back-edges, it will use the bindings for those. If it is not the final activation in a sub-query, it will always produce one binding. If it is on the boundary for any binding it used, then this binding does not need to be propagated to the sub-queries. Thus, the size and composition of the binding environment can change very much between successive activations and can directly affect the average size of a query message by minimizing the number of bindings that it must convey. This is important since most variables will have a scope contained by the clause in which they appear. This is akin to the use of a local stack in most sequential Prolog implementations to handle 'local' variables – variables that only appear within a clause body that can be garbage-collected after the clause is satisfied [MELL82].

2.3. Activity Names

By now it should be clear that each AG vertex can be receiving and sending many messages from and to many other vertices. Hence, some method is needed to distinguish between the various messages as they travel through the network. This can be solved by using the same principles employed in general dataflow systems

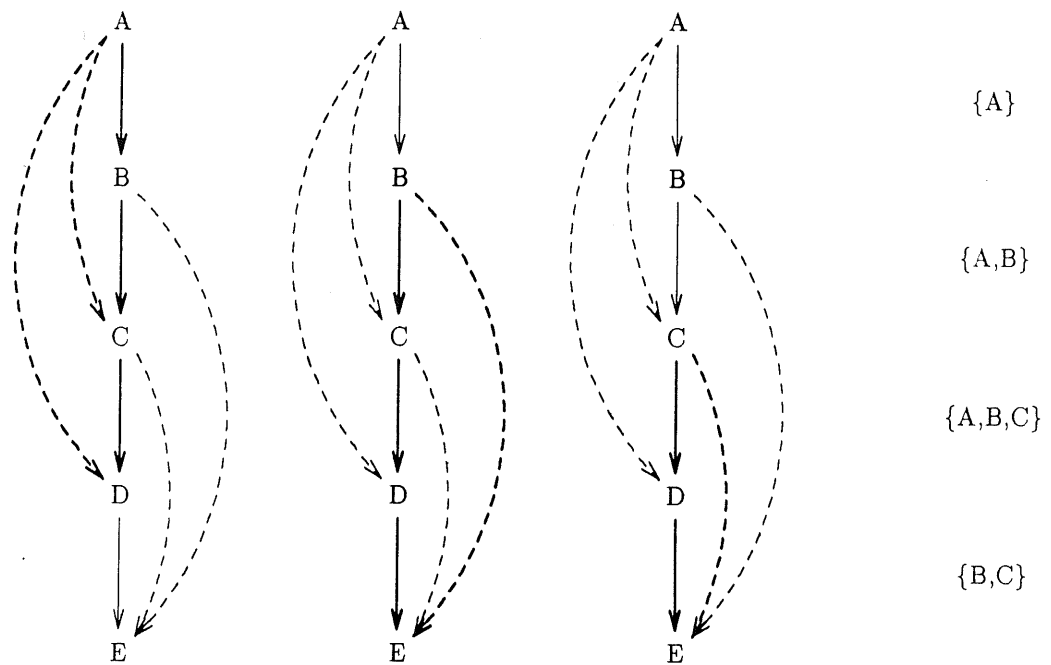


Figure 31

Example of Binding Scope

at the farthest points where they are consumed. In terms of a spanning tree, a variable's scope starts at the vertex where its binding is produced and ends at every descendant to which it is connected via a back-edge. When a back-edge exists, the PE binding to the variable must send the binding information to the descendant at the other end of the back-edge *over the tree-edges* such that the PEs that bind to the descendant (there may be more than one) can check for the existence of the proper back-edge. If no back-edges exist, no descendants need the binding information and, hence, it need not be included in any message.

Example: Figure 31 shows a spanning tree for the query

$$\leftarrow p(A, B)[p(B, C)[p(A, C)|p(C, D)[p(A, D)|p(D, E)[p(B, E)|p(C, E)]]].$$

Tree-edges are solid lines and back-edges are dashed, as usual. The scope of variables, however, is shown by the heaviness of the line or arc. The tree at the far left shows the scope of the variable A . This extends from A itself to D , the

Bounded activity names can be accomplished simply by associating a counter (of bounded length) with each logical vertex. Every time a vertex emits a message, the message is stamped with the name of the sending vertex and the value of the counter which is then incremented. This is sufficient to uniquely identify every message generated during the solution of a query. Of course, bounding the length of the counter bounds the number of outstanding messages any one logical vertex can have without risking the possibility of confusing one message for another. All machines are finite, however, in some respect, so the counter length would just have to be set at some value that is commensurate with other limitations such as total storage. The main attraction of bounded activity names is that they cannot cause the overall message length to grow indefinitely.

Unbounded activity names can be accomplished by concatenating characters from some alphabet to form a string that describes the entire path of activations from the initial injection point to the current logical vertex. This scheme is described in more detail by Bic [Bic84]. While unbounded activity names can cause the overall message length to grow indefinitely, they do provide the following important capability.

It is common knowledge that if one literal in a conjunction is false then the entire conjunction is false. Once one literal is shown to be false then the other literals need not be evaluated. If their evaluations have already begun then they can be canceled. The dual of this is that if one literal in a disjunction is true then the entire disjunction is true. If it is only important to know that there exists at least one solution, then when one literal is shown to be true, the others need not be evaluated. If their evaluations have already begun then they too can be canceled. To do this, we must cancel all subsequent activations from a literal that may have spread to any or all PEs in the network. This may be accomplished by a *global cancellation* mechanism which is facilitated by unbounded activity

[AGP78]. Each message, in addition to carrying the necessary binding information, contains a unique identifier called an *activity name*.

The basic principles governing the generation and use of activity names are as follows. There are basically two types of messages in our system as already mentioned. *Query messages* propagate forward in an attempt to find a match for the graph templates they carry. *Reply messages*, on the other hand, return along the same paths in the opposite direction and report the bindings made during the forward propagation. Whenever a query message is propagated forward, new activity names are generated for the subsequent query messages that initiate work on the sub-queries. Whenever a reply message is propagated backward, the activity name is used to match it to other OR and AND branches of the same parent activation. Another way of thinking about it is that activity names are used to connect different parts of the activation tree as it is built-up in a distributed manner; each activity name uniquely identifying a link over the network.

To be more specific, activity names are used in the following way. Assume that a vertex w_0 has just received a query message conceptually carrying a graph template that has descendent tree-edges e_1, \dots, e_n from w_0 and has an activity name we shall denote as \bar{a} . For each label e_i that matches one or more edges in the assertion graph, w_0 will send a new query message along each such matching edge. These messages will be given unique activity names $\bar{a}_{e_i1}, \bar{a}_{e_i2}, \dots, \bar{a}_{e_ik}$, given k matches for e_i . All these activities are recorded by the vertex w_0 as pending, that is, reply messages with matching activity names are expected to arrive. Note that clause invocation does not directly require any new activity names since it is a local operation. A unique activity name is required only when a literal is matched to the AG and a query message is generated. There are two general methods for generating activity names; one where the length of the name is *bounded* and the other where the length is *unbounded*.

activation environment is a QT tuple for the template and vertex to match and a binding environment that contains *envlen* bindings. Reply messages are divided into *success* and *failure* messages. Success messages contain a tree of successful bindings whereas a failure message contains no such information. The special *cut* message is sent up the activation tree to remove choice points (opportunities for OR-parallel execution) when the cut predicate '!' is encountered.

2.5. Procedures

Finally, we outline procedures for processing these messages in a high-level, pseudo-language. These procedures "summarize" the operational semantics since a precise definition, i.e., complete procedures, would be too lengthy to include here. Figure 33 gives the query message procedure and is the most interesting. Note that the new activity must be processed along with the unstacked activities from the query message as long as head terms in an activity are being matched. That is to say, if the current vertex matches a head term then it must continue processing the parent clause. Also note that only tree-edges or clause invocations can continue forward propagation of messages; the others only specify local operations. Hence, if there are no outstanding sub-queries, then a reply must be sent which may be success or failure. Figure 34 gives the procedures for the success, failure and cut messages. The success and failure procedures are very similar. The main difference is that the success message has bindings that become part of the local environment. Then, success or failure is propagated up the activation tree as long as the current sub-tree has completed, i.e., all sub-queries have finished and all replies have returned. When the root (of the sub-tree at this vertex) is reached, then a reply message is sent to the parent activation. The cut message processing is relatively simple. It propagates up the activation tree setting pointers to NULL that effectively throws-away the alternatives to literals between the cut and the

$\langle \text{msg} \rangle$::=	$\langle \text{msg_header} \rangle \langle \text{msg_body} \rangle$
$\langle \text{msg_header} \rangle$::=	$\text{activity_name src_addr dest_addr}$
$\langle \text{msg_body} \rangle$::=	$q\langle \text{exec_env_stk} \rangle \mid s\langle \text{binding_tree} \rangle \mid f \mid c$
$\langle \text{exec_env_stk} \rangle$::=	$\{\langle \text{exec_env} \rangle\}^+$
$\langle \text{exec_env} \rangle$::=	$\text{body_index vertex_index} \langle \text{binding_env} \rangle$
$\langle \text{binding_env} \rangle$::=	$\text{envlen} \{ \langle \text{binding} \rangle \}^{\text{envlen}}$
$\langle \text{binding_tree} \rangle$::=	$\{ \{ \langle \text{binding} \rangle \}^+ \langle \text{sub_tree} \rangle \}^+$
$\langle \text{sub_tree} \rangle$::=	$(\langle \text{binding_tree} \rangle \mid \epsilon)$
$\langle \text{binding} \rangle$::=	constant variable

Figure 32

Message Grammar

names. To cancel the evaluation of a literal with activity name α , all PEs must asynchronously cancel all activations with activity names $\alpha\beta$ where $|\beta| \geq 0$, i.e., all activations whose activity names have α as a prefix. This information could be “broadcast” to all PEs such that the cancellation occurs promptly. With bounded activity names, on the other hand, cancellation messages would have to be sent individually to “chase” after irrelevant branches of computation. Furthermore, additional work would be wasted until the cancellation messages caught-up with the leaves of the activation tree; assuming that the cancellation messages would be propagated faster than the leaves of the activation tree are extended.

2.4. Message Grammar

All of the concepts presented in the preceding sections can be used to define a message grammar; a precise definition of the information that must be carried in a message and its structure. This appears in Figure 32. A message is divided into a header and a body. The header contains the addresses of the source and destination PEs and the activity name. The message type is defined by the body. Query messages are prefixed with ‘q’ and contain a stack of *activation environments*. Each

Success Message Procedure:

Store bindings in proper environment;
Set success for this sub-query;
if all replies have arrived for this activation **then**
 recursively propagate success up the activation tree for completed branches.
 if root reached **then** send success reply with all bindings to parent;

Failure Message Procedure:

Set failure for this sub-query;
if all replies have arrived for this activation **then**
 recursively propagate failure up the activation tree for completed branches;
 if root reached **then** send failure reply to parent.

Cut Message Procedure:

Set body literal alternatives to NULL;
if clause invocation occurred at this vertex
 then set clause head alternatives to NULL;
 else relay cut message to parent.

Figure 34

Success, Failure and Cut Message Procedures

3. Scheduling

In a tightly-coupled, shared-memory system, any PE can service any memory regardless of how the program of available work (activities) are distributed. Deciding which activity gets done by which PE requires some form of *scheduling*. Since we wish to avoid any kind of central control that would inhibit massive parallelism, we reject any kind of global scheduler. In a loosely-coupled network, we are able to employ *local* scheduling.

The simplest form of scheduling is to maintain incoming messages (available work) in a FIFO queue. Hence, work is processed in the order of arrival and does not require any other knowledge of the computation. Of course, other scheduling policies are possible. In fact, messages can be processed in any order since they do not require any other information. As will be mentioned in Section 6.1, the

Query Message Procedure: (indentation indicates scope of statements)

```

Initialize new activity;
Unstack activations;
for each activation
  if binding doesn't match me
    then set failure and break;
    else bind me to it;
if success so far then
  for each activation
    for each edge
      case edge type of
        tree-edge:
          build new query message;
          send on each matching edge;
          if no matching edges then set failure;
        clause invocation:
          for each matching clause head
            extend the activation tree;
            recursively process edges;
          if no matching clause heads then set failure;
        back-edge:
          match back-edge neighbor to binding in local environment;
          if matching fails then set failure;
        attribute:
          match or set local attribute;
          if matching fails then set failure;
        evaluable predicate:
          look-up arguments in local environment;
          do operation;
          if operation fails then set failure;
      if failure or not at head term then break;
if no active sub-queries then send reply to parent (success or failure).

```

Figure 33

Query Message Procedure

clause head. The cut message is relayed to parents until the vertex is reached that actually did the clause invocation.

As noted in Chapter 2, static allocation cannot utilize PEs that become idle and cannot alleviate hot-spots. In BPEM, however, we note that multiple top-level queries can be executed in parallel since they are logically independent. These queries could be utilizing different parts of the AG and, hence, increasing the overall network utilization. Also, in the next section, we will discuss a form of dynamic load-balancing using a hybrid architecture.

As also noted in Chapter 2, the problem of mapping one graph on to another such that some metric is optimized is, in general, NP-Complete. Hence, a practical method of finding good, sub-optimal solutions must be found. Such methods usually exploit some special case or property of the problem graph. Approaches like these are not likely in BPEM since an AG is not a dataflow or task-precedence graph; it is a *task-interaction graph*. An AG does not determine when or under what conditions two vertices will communicate or in which direction; it only says who *may* communicate. QTs are also not dataflow graphs since each sub-tree in a QT may have multiple instances in an AG. Also, an AG may be very irregular such that there may be no regularities that can be exploited for mapping.

4.2. Simulated Annealing

Because of these limitations, we will use the more general approximation technique of *simulated annealing*. This technique was introduced by Kirkpatrick, Gelatt and Vecchi [KGV83] and has been used for a wide variety of NP-Complete combinatorial optimization problems [KGV83, RVS84, MOLI85, IW87] in addition to Boltzmann machines [FHS83, AHS85]. Simulated annealing is essentially an iterative improvement algorithm but it seeks to avoid entrapment in a local minima by probabilistically accepting “uphill” moves.

In engineering sciences, annealing is the process of heating and gradually cooling materials, such as glass or metal, to eliminate internal stresses. *Simulated annealing* is patterned after the physical annealing process as a close analogy. It

scheduling policy or queuing discipline can be used to control the generation of parallel activities depending on how full the input queue is. A priority queue could also be used based on the sender of the message or how much work a message represents.

4. Allocation

As mentioned before, if the AG is considered to be an active network of PEs, there is no *allocation* of program and available work that needs to be done. Each vertex must handle its own incoming messages. To build a network in the shape of an AG, however, is not practical. Every AG can have a different irregular shape while a physical network will have a relatively small, fixed number of PEs in a very regular shape. Hence, the *logical* architecture of the AG must be allocated or *mapped* on to the *physical* architecture of the network.

4.1. Static vs. Dynamic Allocation

Any such map must satisfy two competing goals as well as possible:

- 1) even distribution of work, and
- 2) minimized communication distances.

The obvious mapping that satisfies (2) maps all AG vertices on to one PE. The communication distance is zero but this is the worst possible distribution of work. It is also possible to satisfy (1) but every communicating pair of AG vertices may be mapped to diametrically opposite sides of the network.

The even distribution of work, or *load balancing*, is difficult to do since the amount of available work is constantly changing over the course of a computation. Any kind of *dynamic* allocation at run-time is especially difficult since this requires the overhead of moving an AG vertex from one PE to another along with all incident AG edges and possibly forwarding any outstanding messages. Hence, due to its simplicity, we will use *static* allocation.

a problem of size n , “equilibrium” is reached after an changes have been accepted or bn changes (accepted or not) have been attempted, whichever comes first; a and b are chosen such that $1 \leq a < b$. It should, however, be pointed out that this equilibrium criterion is only a simplification, since, in general, the number of moves necessary to reach equilibrium increases as the temperature is lowered.

After equilibrium is reached, the temperature is decreased, and the same process of random changes is repeated until equilibrium is reached again. The lowering of the temperature typically follows an exponential decay function until the system is considered “frozen” and annealing terminates. We use the simple termination condition cited in [RVS84] which declares the system frozen when E has not changed for three consecutive temperatures.

The main disadvantage of simulated annealing is that it can be slow for larger problems. We can use a faster special case of annealing that is called *quenching*. This corresponds to a rapid cooling of the system to a freezing temperature. Note that when $T = 0$, all decreases in energy are accepted and all increases are rejected. Hence, quenching approaches *iterative improvement*. While the ability to avoid local minima is decreased, experience has shown us that the solutions found by quenching are only slightly worse but take a small fraction of the time.

4.3. The Distance-Variance Energy Function

To use simulated annealing on the mapping problem, we must define some energy function such that given a network graph $G_n = \langle V_n, E_n \rangle$ and an assertion (problem) graph $G_p = \langle V_p, E_p \rangle$, it returns the energy value E for any mapping $m : V_p \mapsto V_n$. (Note that we are particularly interested in the case where $|V_n| \ll |V_p|$.) As mentioned before, E must measure the competing goals of minimizing the communication distance and evenly distributing the work across the entire network.

is applicable to any combinatorial optimization problem whose solution can be represented by a configuration or set of values in which point-wise changes can be made. In physical annealing, the *energy* of the system is determined by its *temperature*. In simulated annealing, “temperature” is a parameter defined by the user. Similarly, the “energy” of the system is determined by a user-defined *energy function* (also called the *cost* or *objective function*). The “energy” E at every point in the computation represents the “goodness” of the current solution.

The simulated annealing process begins at a high “temperature” level by performing a series of random changes to the current solution, just as random changes would occur in the physical system. After each change, the resulting “energy” E' of the configuration is calculated. The change $\Delta E = E' - E$ corresponds to improving or worsening the solution. This change is accepted and becomes part of the solution with probability

$$P(\Delta E) = \frac{1}{1 + e^{\Delta E/T}}$$

where a decrease in the energy (negative ΔE) has an increasing probability and an increase (positive ΔE) has a decreasing probability. We will vary this decision criterion by accepting all decreases in energy while probabilistically accepting increases. This probability can be evaluated as

$$P(\Delta E) = e^{-\Delta E/T} \quad \text{for} \quad 0 \leq \Delta E.$$

Note that in both cases when the temperature is high, gross changes in E may occur. As the temperature is decreased, parts of the solution are “fine-tuned” as it settles down (hopefully) close to a global minimum, i.e., an optimal solution.

The series of random changes continues until the systems reaches “thermal equilibrium”. To determine when this occurs, however, is not a simple manner. For the purposes of this paper, we used the same conditions proposed in [KGV83]: For

problem vertices during the optimization process do not alter the distribution of the problem (the LV metric) and hence the minimization function uses only the QD metric (or the cardinality as a special case) to derive a solution.

In the general case, however, we must first scale the two metrics to the same numerical range before they can be weighted. To do this, we will identify the minimum, maximum and *mode* values for QD and LV . The minimum QD occurs when all AG edges are mapped to one PE. The maximum QD occurs when all communicating AG vertices are mapped to diametrically opposite sides of the network. Hence, we have

$$QD_{min} = 0, \quad QD_{max} = |E_p| \cdot diam$$

where $diam$ is the diameter of G_n , i.e., the network. The mode of the distribution of the quadratic distance for all possible maps, however, occurs when each AG edge is mapped only half-way across the network:

$$QD_{mode} = \frac{|E_p| \cdot diam}{2}$$

The minimum LV occurs when there is a perfect distribution. The maximum LV occurs when all AG edges are mapped to one PE. Hence, we have

$$LV_{min} = 0$$

$$LV_{max} = \frac{(|V_n| - 1)d_{avg}^2 + (2 \cdot |E_p| - d_{avg})^2}{|V_n|} = 4 \cdot |E_p|^2 \cdot \left(\frac{|V_n| - 1}{|V_n|^2} \right)$$

where

$$d_{avg} = \frac{2 \cdot |E_p|}{|V_n|}$$

is the average logical degree for any PE. The mode of the distribution of the logical variance, however, does not fall in the middle of this range; it is extremely skewed towards LV_{min} . LV_{mode} occurs when the logical degree varies from the average by one half. Hence,

$$LV_{mode} = \frac{\Sigma \left(\frac{3 \cdot d_{avg}}{2} - d_{avg} \right)^2}{|V_n|} = \frac{|E_p|^2}{4 \cdot |V_n|^2}$$

To minimize the communication distance, adjacent vertices in the problem graph must be mapped as close together in the network as possible. Hence, the sum of the network distances between any two adjacent problem vertices,

$$QD = \sum_{(v_1, v_2) \in E_p} \text{dist}(m(v_1), m(v_2))$$

where *dist* is the network distance between any two PEs, must be minimized. This is called *QD* for the *quadratic distance* due to its similarity with the quadratic assignment problem. (Note that since an attribute is owned by one AG vertex, it will be kept local to that vertex, i.e., it will simply follow the mapping of its owner and not enter into the annealing process.)

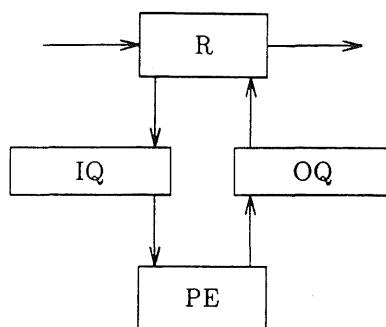
For most applications, it is desirable that work be evenly distributed throughout the network. In the absence of any prior information about the application, we assume that all problem edges carry an equal amount of traffic and the processing of each message generates an equal amount of work. Under this assumption, an even distribution of the problem graph is achieved by distributing the problem edges such that the variance of the number of problem edges incident on each PE is minimized. Hence, we minimize *LV*, defined as:

$$LV = \text{var}(LD_w), \quad \text{for } w \in V_n, \quad \text{where}$$

$$LD_w = \sum_{v \in \text{inverse-image}(w)} \text{deg}(v), \quad \text{for } v \in V_p, w \in V_n.$$

LV is called the *logical variance* of *LD*, the *logical degree*, which is the number of logical edges incident on each $w \in V_n$.

As mentioned above, these two metrics are competing and work in opposite directions. To find useful maps, we must use a weighted composite of these two metrics as the energy function. In systems where $|V_p| \leq |V_n|$, *LV* can be simplified by requiring that each problem vertex is mapped onto a different PE, i.e., that *m* be one-to-one, as in the approach described in [BOK81]. Pairwise exchanges of

**Figure 35**

A PE with Separate Router

us to *hybrid* architectures where locally a tightly-coupled, shared-memory design is used but the overall global design is still a loosely-coupled, message-passing network.

5.1. Routing

The first step is to separate the message routing function from those functions directly related to BPEM. A dedicated router or “post office” can be used to send and receive all messages on the channels to and from the immediate neighbors as shown in Figure 35. If a message is addressed to the router’s PE, then it is placed in the PE’s input message queue rather than being sent on the next hop of its path. All output messages that a PE generates from an input message are placed in the output queue. Thus, a PE only has to deal with its own messages and does not have to deal with routing. The amount of time that is saved, however, depends not only on the amount of traffic through a particular router but also on the average number of hops per message. If a parallel programming paradigm only requires an average number of hops of about 1.0, then independent routers would not save any time. This may not be that uncommon since any parallel system needs to minimize the communication overhead or distance. This is, in fact, the case for BPEM as shown in Chapter 6.

Using these mode values, we can scale the metrics and weight them:

$$E = W_{qd} \frac{QD}{QD_{mode}} + W_{lv} \frac{LV}{LV_{mode}}$$

where W_{qd} and W_{lv} are the weights and

$$0 \leq W_{qd}, W_{lv} \leq 1 \quad \text{and} \quad W_{qd} + W_{lv} = 1 .$$

4.4. NP-Completeness of the Mapping Problem

To show that the mapping problem in its general form is NP-complete, we can restate it as a decision problem. Choose a value J for the quadratic distance and a value K for the logical variance and ask: Is there a mapping such that $QD \leq J$ and $LV \leq K$? This is a clear instance of the Graph Partitioning problem which is NP-Complete. (See [GJ79], problem [ND14].) Each PE represents a partition and mapping a logical vertex is assignment to a partition. This fits well with our assumption that the problem size is in general larger than the network size.

5. Supporting Architectures

Simulated annealing allows us to map an AG onto any network. Hence, the only topological requirements of the network are the fundamental ones of degree, diameter, symmetry and scalability. As reviewed in Chapter 2, there is a wide variety of topologies that have been proposed. In the subsequent chapter on simulation results, we will use n -cubes because of the low diameter and easy routing and also because of its wide popularity.

In all previous discussions on loosely-coupled, message-passing networks, however, we tacitly assumed that a PE is an atomic entity with private memory that can communicate with neighbors over some number of channels. In this section, we discard this assumption and examine the structure of a PE to understand how parallelism can be exploited from *within* a PE on a hardware level. This will lead

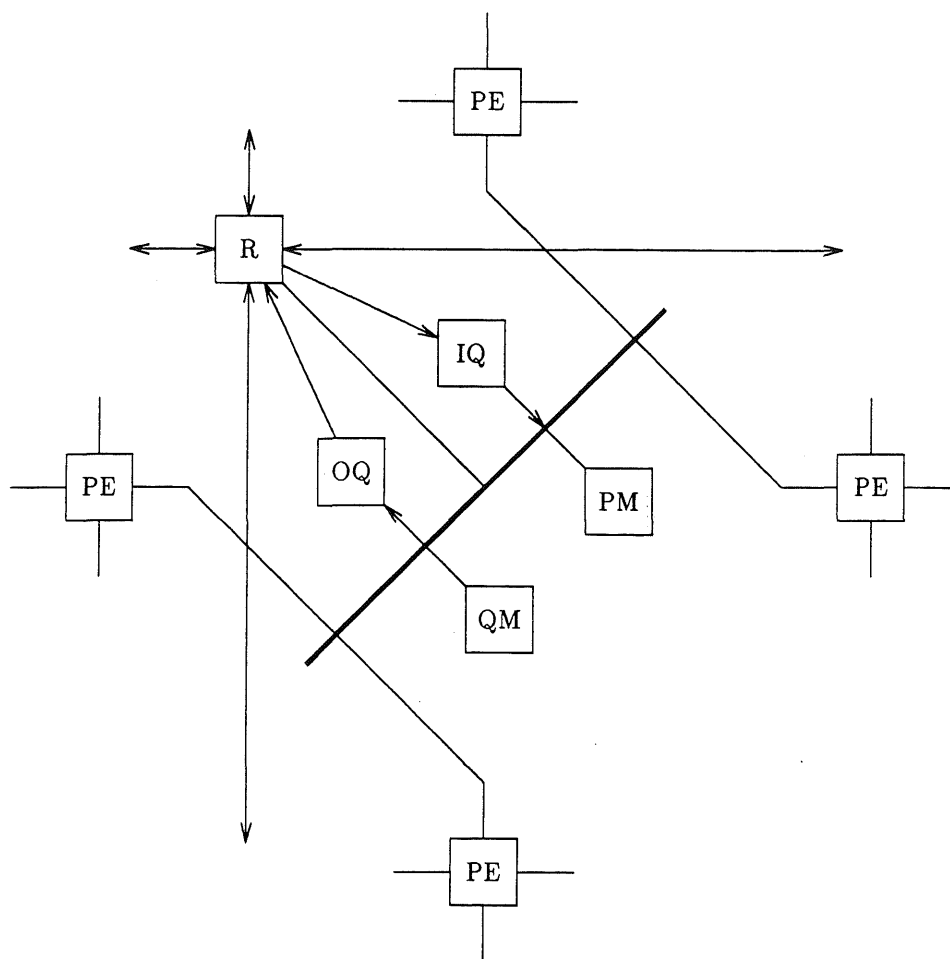


Figure 37

Variable Service

of the same activation. It would be possible, however, to implement the QM as a *segmented* or *interleaved* memory and to assign activations to different sections of memory. Contention for the PM, on the other hand, is much less since the PM is written only at initialization (possibly through the router) and when an attribute value is assigned. Since attributes are logically immediate neighbors, it is convenient to consistently store this information in the PM. If attributes were stored in the QM, however, the PEs could treat the PM as a read-only memory.

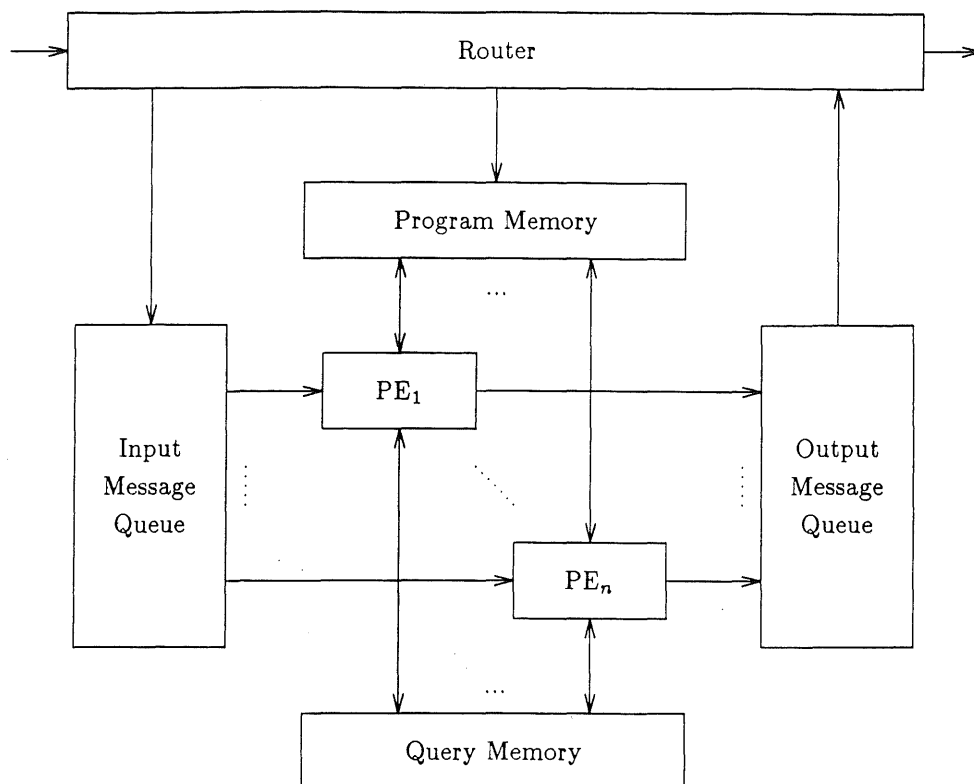


Figure 36

Multiple Message-servers

5.2. Multiple Message-servers

The structure of a PE can be further refined as in Figure 36. Since a PE can be thought of as a *message-server* for a queue of input messages, we can certainly have *multiple* servers to increase throughput. We can also refine the memory functions. The Query Memory (QM) contains dynamic data, i.e., activations that are built and dismantled as query and reply messages are processed. The Program Memory (PM) contains “static” data, i.e., the QTs and the attributes and logical neighbors for each AG vertex mapped to this PE.

With multiple PEs, both the QM and PM are *shared-memories*. Hence, memory access must be arbitrated to avoid corruption due to parallel writes. In the QM, this is only a problem when two PEs want to process the same vertex

6.1. Queue Overflow, Deadlock and Resource-Driven Parallelism

In this chapter and in the simulations reported in the next chapter, we have assumed that the message queues are infinite in length. Hence, we do not worry about what happens when a queue fills-up. In a physical machine, however, overflow is a serious situation. If full input queues were able to block message transmission from neighbors, then deadlock would be possible. This would occur if two neighboring PEs with full input queues attempted to send messages to each other.

This brings up the question of controlling or throttling parallelism at the message level. Clearly, unbridled parallelism that can swamp a machine is not desirable. Since every machine is finite, the size should be able to affect the amount of active parallelism. This is the concept of *resource-driven* parallelism. When the PEs become saturated, the spawning of new parallel activities is reduced until the load decreases. One method to accomplish this is to have PEs switch from parallel (breadth-first) to sequential (depth-first) execution as their queues fill-up. Another method is to switch the queue discipline from FIFO to LIFO. This does not cause strict depth-first exploration of the search space but rather causes the activation tree to be expanded at deeper and deeper levels such that the leaves of the search space are (hopefully) reached before overflow occurs [WATS88]. This can be thought of as *sparse breadth-first* execution.

In general, PEs under resource-driven parallelism must be able to efficiently execute in parallel and in sequence. If there are sufficient PEs, then they must execute code in parallel. If they are fully-loaded, they must execute code sequentially. They must also efficiently record the opportunities for parallelism such that these records don't swamp the machine. While resource-driven parallelism is a very interesting idea, it will unfortunately not be pursued further in this thesis.

5.3. Variable Service

We can extend the concept of multiple message-servers to include a form of *load balancing*. Load balancing is usually done by sending work to less busy PEs. The dual of this is skewing a PEs service to those input message queues that are more busy. This duality can be stated in a larger context. All computation is the *application* of an *operation* to *data*. Application can only occur when the operation and data are connected in some sense. Classical dataflow supports parallelism by sending the data to the operation. The dual of this is sending the operation to the data.

Consider Figure 37 which, for sake of argument and without loss of generality, assumes a grid topology. The central feature here is a bus represented by the heavy diagonal line. The router handles communications, as usual, and places messages in the input queue. Each of four different PEs, however, can service this queue and interact with the PM, QM and the output queue. Note that each PE in the network can talk with a different subset of buses and their memories. If all input queues are equally loaded, then all PEs equally divide their service between the buses in their subset. If an input queue is more loaded, then the PEs will give it a larger share of their time. This requires, of course, some *policy* for allocation of service. While this cannot eliminate hot-spots, it can alleviate them by providing variable service. Given the previously mentioned difficulties in load-balancing a loosely-coupled, message-passing system where a large amount of data (the AG) is statically allocated, this might be a viable alternative.

6. Other Issues

We conclude this chapter by discussing several issues which would not directly affect BPEMs viability but would have to be addressed in any genuine implementation.

CHAPTER 6

Simulations Results

1. Introduction

In this chapter, we present simulation results that demonstrate the model's fundamental performance capabilities. We will do this by using problems that exercise very specific aspects of the model, e.g., OR-parallelism, AND-parallelism and clause invocation. This will give us an estimate of the model's peak performance under each type of operation. The average performance as exhibited by "real-world" problems will, of course, depend on the mix of these operations and primarily on the depth of clause invocation. In the absence of comparable performance data for "real-world" problems, an estimate of performance in absolute terms (logical inferences per second, lips) will demonstrate the model's viability:

Besides giving us an idea of the model's peak performance, using specific problems will allow us to demonstrate a very important property: the processing of OR-parallelism, AND-parallelism and clause invocation is essentially linear such that the model's performance will scale-up; a larger machine will run a proportionally larger problem with similar performance. There are, however, cases where this property is not maintained. When clause invocation is nested, as would occur in recursion, the processing of each level of invocation becomes more expensive and performance suffers. This will be demonstrated.

The following section describes the simulator and various metrics that can be used to evaluate performance. Section 3 then demonstrates that performance under OR-parallelism scales-up. Section 4 demonstrates that AND-parallelism is handled just as efficiently as OR-parallelism. Section 5 demonstrates clause invocation.

6.2. Adaptive Routing

Routers can become hot-spots as well as PEs or message queues. If messages from PE_i to PE_j are always sent over the same path, then a combination of such paths may cause some routers to become overloaded. In any network, there may be one unique path that is the shortest. In general, however, there are many paths that are not much longer. *Adaptive routing* can increase overall message throughput by finding alternate, if slightly longer, paths around overloaded routers.

6.3. Fault Tolerance

While there are many possible failures in a complex system, we will only briefly discuss three broad categories: (1) communication channels, (2) PEs, and (3) memories. The methods of dealing with these faults are based on the ideas of *redundancy* or *shared resource*.

Communication channel faults can be tolerated by using adaptive routing. If a channel fails, adaptive routing could treat this as a severely overloaded router and simply attempt to re-route the message around that channel. PE faults can be tolerated using the multiple-server schemes of Figures 36 and 37. If any one PE failed, the system would continue to function, albeit a little slower. Memory faults can be tolerated by using error detecting/correcting memories. Here, redundancy (extra bits) must be built into each word since the AG vertex information is only loaded into one PM and the activations are only built in one QM.

list of queries for simulation. These are written in Restricted Binary Logic (RBL). The fourth file (S.A. Prm.) contains network information and simulated annealing parameters.

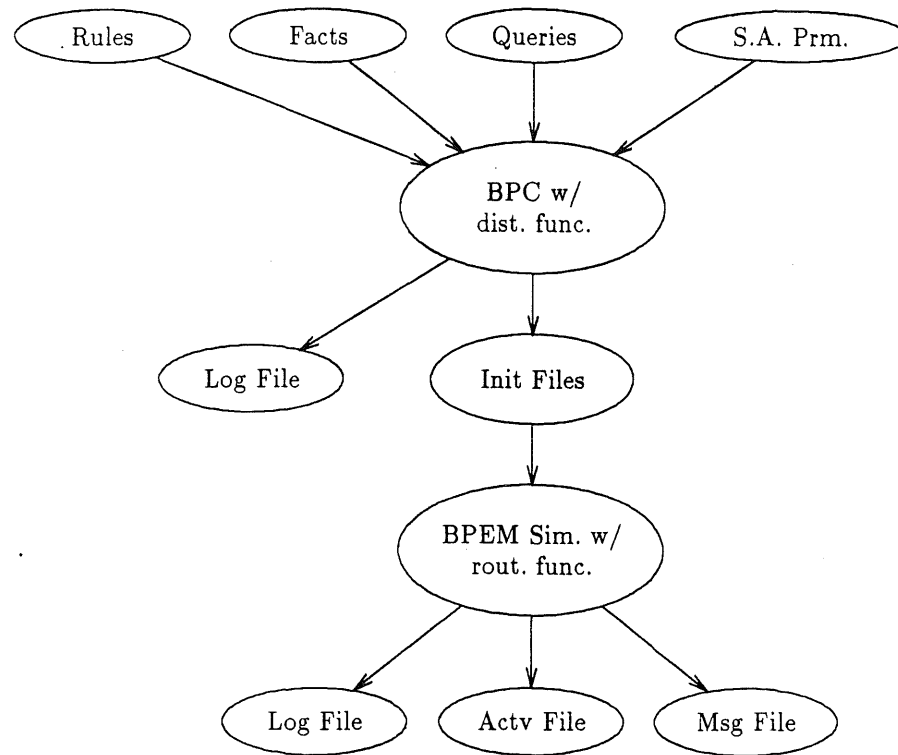
The compiler first parses the facts files and builds the Assertion Graph. The parameter file is read and the AG is annealed on to the network graph as discussed in Chapter 5. Specifically, the parameter file gives

- 1) the number of PEs in the network,
- 2) the diameter of the network,
- 3) the quadratic distance weighting factor,
- 4) the logical variance weighting factor,
- 5) the initial temperature,
- 6) the temperature decay factor,
- 7) the maximum-number-of-moves factor a , and
- 8) the maximum-number-of-attempted-moves factor b .

In addition, a distance function (dist. func.) for the network topology being simulated is linked into BPC. All of this information is used to evaluate the energy function and conduct the annealing.

Once this is done, the rules and queries are parsed. This is very similar since most of the work is involved in processing the clause bodies. This requires (1) building a spanning tree for each clause body (rule or query), (2) identifying all occurrences of variables that occur in the head and their branch to the root, (3) finding all producers/consumers of shared variables and their most recent ancestor, and (4) identifying the binding environment that must be sent by a vertex to each of its immediate descendants.

When this is completed, two initialization files are written: `initpe1` and `initpen`. The file `initpe1` is read by a special PE, simply named `PE1`, and uses the information to "inject" queries into the proper AG vertex. `initpe1` also contains a symbol table such that query solutions can be printed in terms of the original query. The file `initpen` is used to initialize all other network PEs. It specifies which PEs

**Figure 38**

Simulator Flow Graph

Section 6 demonstrates performance on a wide range of very small to very large problems and shows the effect on the number of active PEs and the queuing. Conclusions are given in Section 7.

2. The Simulator

These simulations involve two major programs as shown in the simulator flow graph in Figure 38: the Binary Predicate Compiler (BPC) and the Binary Predicate Execution Model Simulator; both written in C. We discuss these separately.

2.1. The Binary Predicate Compiler

BPC requires four input files and produces a log file and the initialization files for the actual simulator. Three of these input files contain the facts, rules and a

- 2) PE active time,
- 3) messages/clause invocations per unit time,
- 4) message queue length at any point in time,
- 5) communication time, and
- 6) average message length.

The overall query execution time is the simulated real-time required to completely solve a query. If we look at the active time for all PEs, then we can derive an activity curve for all PEs throughout a query that describes the PE utilization. Since a message can be considered unification with the AG and invocation is unification with a clause head, the sum of these tell us the total number of unifications or logical inferences per unit time. The length of input message queues can tell us when the network as a whole is becoming overloaded or when a hot-spot is developing. Since each PE is handling message routing in software, the communication time denotes how much time is spent doing it. We can also estimate the goodness of the mapping by the average number of hops each message must take. The average message length, on the other hand, is an approximate measure of how complicated the average message is and how much processing time it will take.

2.3. Timing Instrumentation

Most of the metrics above require some notion of timing the simulation events. The timing provided by the Unix operating system, however, was found to be inadequate. The only resolution available is 10 msec. which is too coarse. When Unix timing was used, many messages would be processed in "zero" time. To avoid this, the assembler code for the simulator that is produced by the C compiler was examined. The number of instructions for each loop and branch were counted. For each procedure, loop and if statement, the simulator keeps a running total of the number of instructions executed and uses this figure as the elapsed running time. This figure is not exact but is closely proportional to the true behavior of the simulator to within a small constant since the time for every procedure, loop and

handle which AG vertices and contain all clause body information, whether it is a query or a rule.

Finally, BPC produces a log file. This identifies which rules, facts and queries were compiled, i.e., which problem resides in the initialization files, and the annealing parameters that were used. The initial and final (annealed) mapping metrics are recorded such that the goodness of the maps and the efficiency of the annealings can be evaluated.

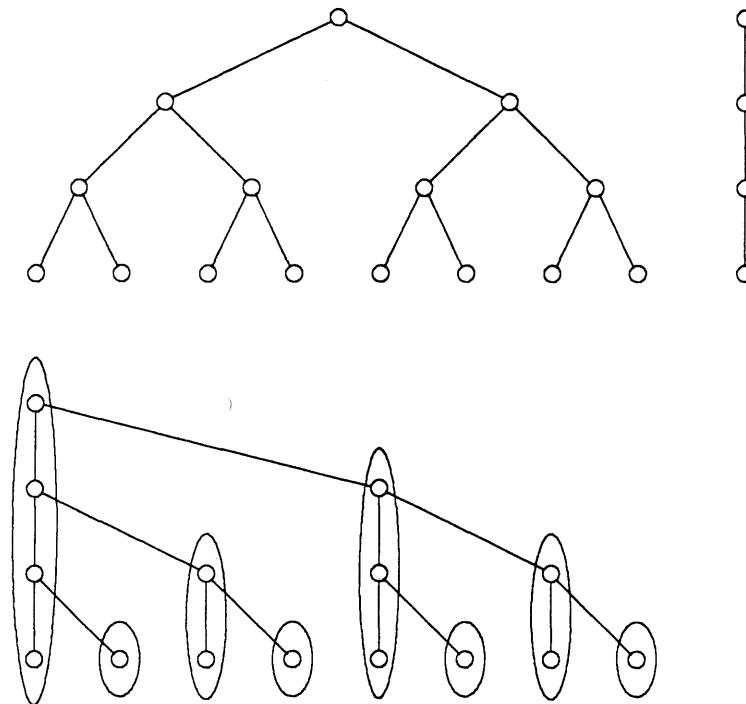
2.2. The BPEM Simulator

The BPEM Simulator is built on top of a modified version of the Marlin Dataflow Emulator [MARL84]. It essentially implements a message-passing star network with the emulator at the hub and the simulated PEs at the end of the spokes. Time-stamped messages are stored in a priority queue. Running on a receive-to-receive cycle, the oldest message is removed and passed to the appropriate PE. In reality, the PE code goes through a hash table to find the appropriate AG information and current activations, i.e., previously processed messages for this PE whose replies are pending. In processing the current message, new query or reply messages could be enqueued with the simulator.

The BPEM simulator produces three files: (1) the log file, (2) the activity file, and (3) the message file. The log file captures information from the simulator itself and also special log messages from the simulated PEs. The answers to initial queries are recorded here as special log messages from PE₁. The activity file records when each PE is busy, when it receives or transmits a message, and if it is just routing a message destined for some other PE. The message file records when a message is sent, how long it is in transit, its length, how long it spends in the queue, and if it is taking the last hop before reaching its final destination.

These files allow each simulation to be evaluated on the following metrics:

- 1) overall query execution time, t_x ,

**Figure 39**

The Binary Tree Search Problem

Using an AG that is a balanced binary tree may seem like a very special case but BPEM does not exploit the properties of balance or having two sub-trees for performance. The AG could be an arbitrary tree or, in fact, an arbitrary graph; this makes no difference in satisfying a query. The benefit that is derived here is that the problem size and depth and the required number of logical inferences are precisely known.

if statement is represented. If we assume each instruction executes in $1 \mu\text{sec.}$, then we can estimate the model's performance in logical inferences per second (lips). In the simulations that follow, we will also assume that messages are transmitted between PEs at $1 \mu\text{sec./character.}$

3. OR-Parallelism

Here we show that BPEM performance scales-up under OR-parallelism. By using a problem that exhibits only OR-parallelism, and keeping a constant proportion between the problem size and network size, we can show similar performance as constrained by the maximum parallelism available in the problem. This demonstrates that overhead costs, such as communication, synchronization, etc., do not grow faster than the problem size. We will show this for three proportions: when the problem size is 4, 8 and 16 times larger than the network. We begin by describing the problem.

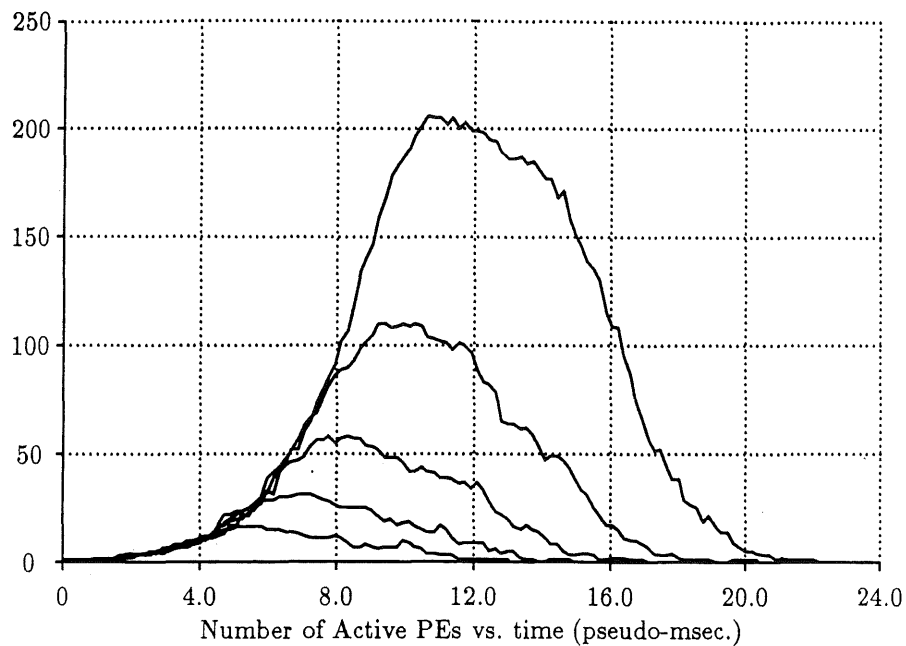
3.1. The Binary Tree Search Problem

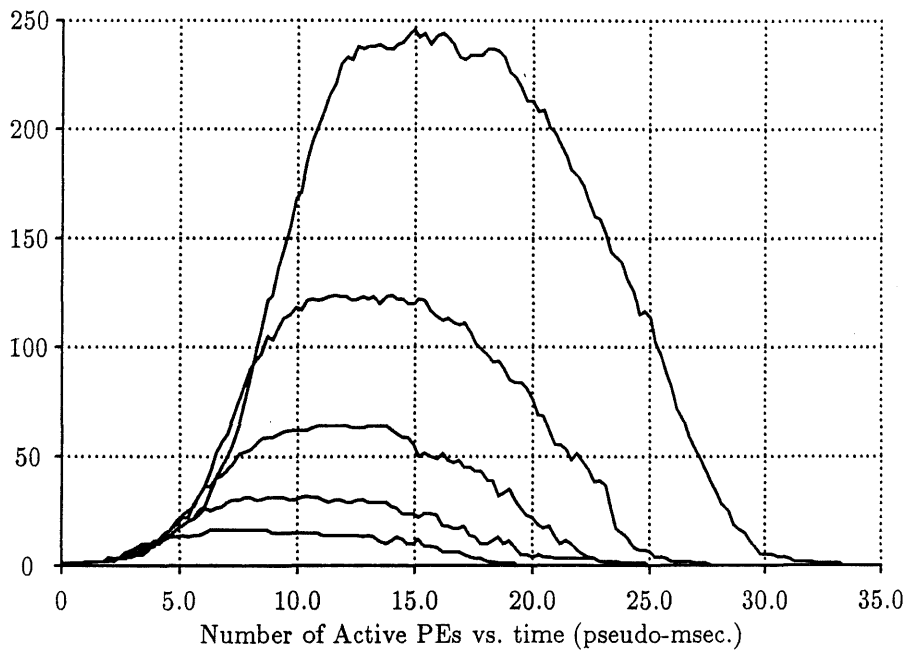
Figure 39 illustrates the Binary Tree Search Problem. The Assertion Graph (AG) is a balanced binary tree and the problem is to find one solution at a leaf. Hence, the query template is a chain, shown at the right of the binary tree, that must match one specific branch. Note that all AG and query template edges can have the same name, hence, they are not labeled here. If at every vertex, one branch is given to another PE, as illustrated by the tree with ellipses in Figure 39, then maximum parallelism is achieved. Given that the tree height is h , we can say

$$\text{Problem Size} = 2^{h+1} - 1$$

$$t_x = O(h)$$

Note that the number of vertices in the AG and the number of query messages (logical inferences) all equal $2^{h+1} - 1$. Also note that even under maximum parallelism, the execution time, t_x , is constrained by the tree height h .

**Figure 41**Activity Curves for 8x Proportion

**Figure 42**Activity Curves for 16x Proportion

h	5	6	7	8	9	10	11
$ AG $	63	127	255	511	1023	2047	4095
4x							
$ PE $	16	32	64	128	256		
t_x	10.64	10.34	13.02	14.93	18.01		
t_x/h	2.13	1.72	1.86	1.87	2.00		
k-lips	5.92	12.28	19.59	34.23	56.80		
8x							
$ PE $		16	32	64	128	256	
t_x		13.05	14.91	17.11	20.41	22.15	
t_x/h		2.11	2.13	2.14	2.27	2.22	
k-lips		9.73	17.08	29.87	50.12	92.42	
16x							
$ PE $			16	32	64	128	256
t_x			19.30	24.86	24.34	27.62	33.28
t_x/h			2.76	3.11	2.70	2.76	3.02
k-lips			13.21	20.56	42.03	74.11	123.05

Table 2

Simulation Results under OR-parallelism

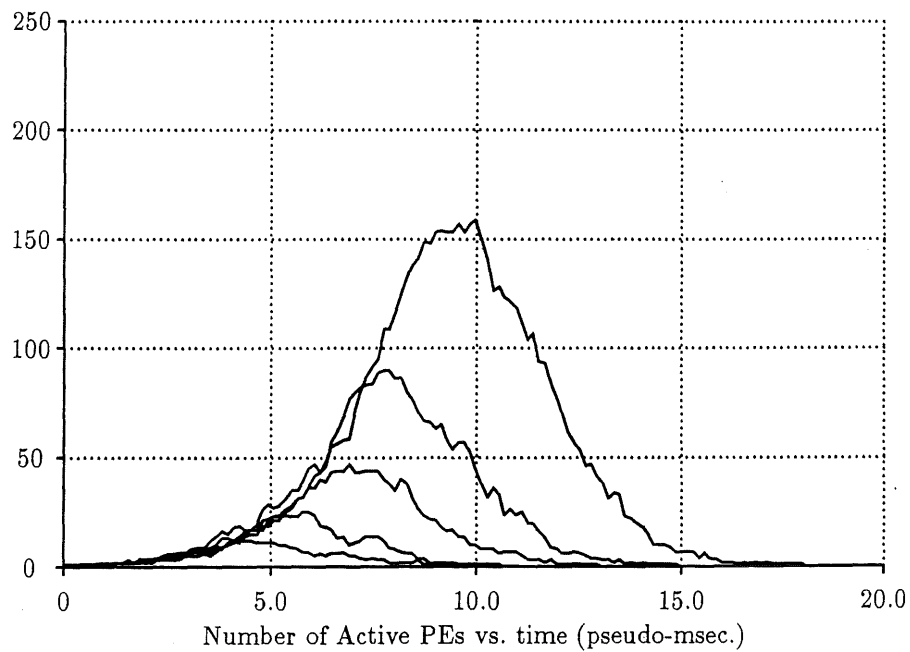


Figure 40

Activity Curves for 4x Proportion

3.2. Results

Here we present simulation results for the Binary Tree Search problem when the AG is 4, 8 and 16 times larger than the network. In all cases, the network is a cube and for each proportion, five values of h are used. The size of the cube is 2^h and its diameter is h . For the annealings, the initial temperature was 100 and the decay factor was 0.9. Since the degree variance of the AG vertices was not great to begin with, a 0.9/0.1 weighting in favor of the distance metric was used. The maximum-number-of-moves factor and the maximum-attempted-number-of-moves factor was 1 and 2, respectively. This made the annealings more like quenching and relatively quick.

Table 2 summarizes the results. For each proportion, the sizes of the AG, $|AG|$, and the network, $|PE|$, are given. Also given are the overall query execution times, t_x , in thousands of time units (instruction counts). As mentioned before, if we assume that each instruction executes in 1 μ sec., then these values can be considered pseudo-milliseconds. Since maximum parallelism is constrained by h , the ratio t_x/h indicates the model's performance relative to the theoretical optimum. Also shown are the number of logical inferences per (pseudo) second in thousands (k-lips). The activity curves (number of active PEs at any given instant) for each proportion are shown in Figures 40, 41 and 42. In each figure, the larger problems bring more PEs into the computation in parallel. The overall execution time, t_x , for each query, however, does not increase in proportion to the problem size but rather in proportion to the problem depth, i.e., h . Hence, the t_x/h ratios are essentially flat as shown in Figure 43. (Some variability can be expected due to the maps produced by simulated annealing.) This shows that the model provides maximum parallelism as the problem size and network size increase. In other words, the model's performance scales-up as constrained by the available parallelism. If we look at the k-lips plotted in Figure 44, we see that the

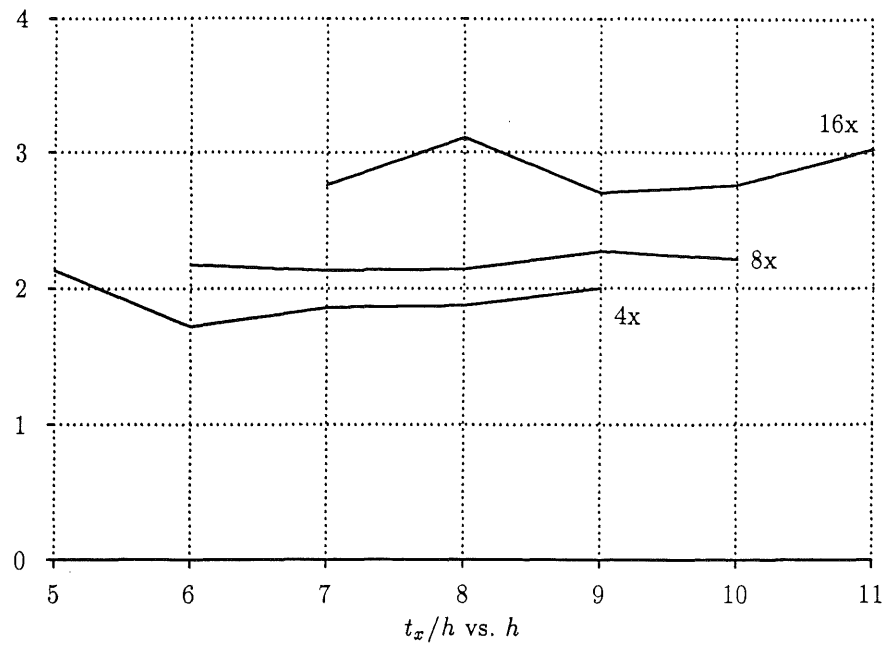


Figure 43

Relative Execution Times for each Proportion

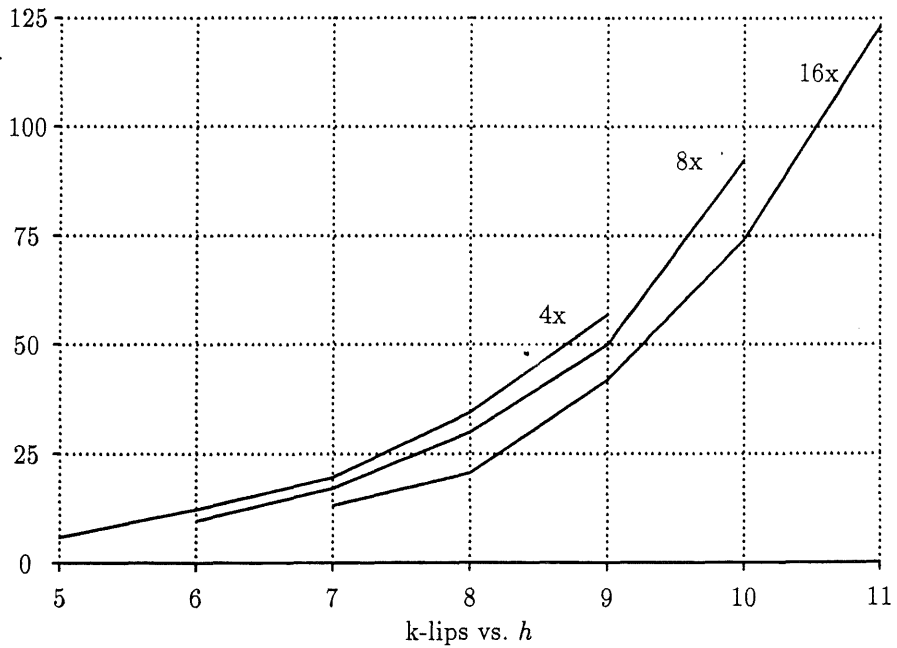


Figure 44

K-lips for each Proportion

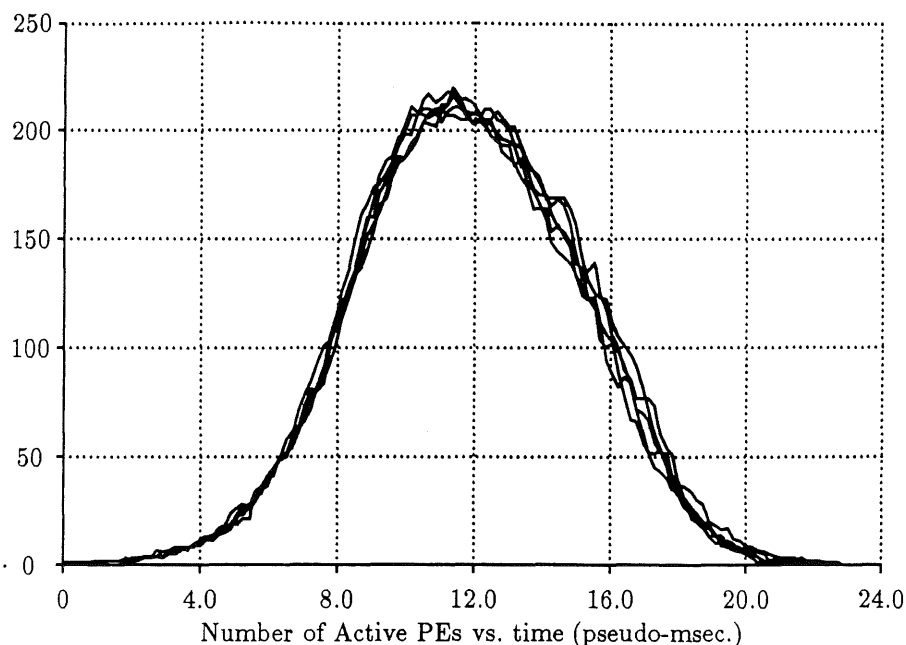


Figure 46

Activity Curves for Different Amounts of AND-parallelism

each pair of AG edges at this level are also given two different labels (such that half are g and half are d as in Figure 45), then we have preserved the same amount of overall work. Since these two edges will be matched at the bottom of the AG, we have in fact replaced 50% of the OR-parallelism with AND-parallelism. If we progressively build up a binary tree from the leaves, then we progressively replace more OR-parallelism following the inverse powers of two. Note that to achieve 100% AND-parallelism, the query template and the AG must be equal in size since replacing each edge in the template with a pair results in a tree that is identical to the AG.

4.2. Results

A series of simulations were done where a constant $h = 10$ was maintained and the height of the AND-parallelism, h_a , was varied from 1 to 5. The resulting percentage of AND-parallelism and the overall query time is given in Table 3. The

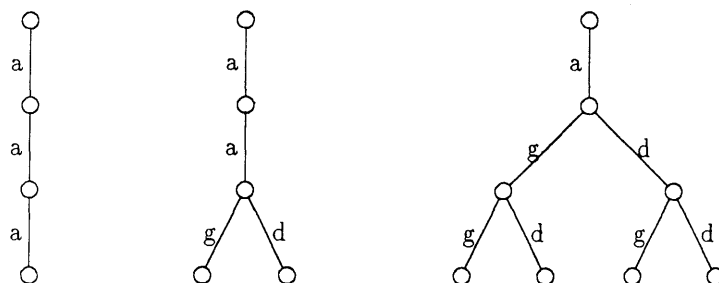


Figure 45

Replacing OR-parallelism with AND-parallelism

larger networks are effectively utilized to obtain more “horsepower”. As long as the networks are not saturated, they can provide exponentially more k -lips for a given problem. As a rough maximum, ≈ 100 k -lips are possible for 256 PEs. Note that for any given h , however, the higher proportions provide slightly lower k -lips. This is due to hot-spots arising as the AG tree becomes larger. A method of dealing with this will be discussed in Section 6. For comparison with later simulations, we note that the average message length for all queries was ≈ 8.5 characters.

4. Introducing AND-Parallelism

In the Section 3, we used the Binary Tree Search problem to demonstrate BPEMs behavior under just OR-parallelism. In this section, we will modify the problem such that the OR-parallelism is progressively converted to AND-parallelism while the total amount of work remains the same. This will show that AND-parallelism is handled just as efficiently as OR-parallelism since the resulting simulations are virtually identical.

4.1. Problem Statement

Recall that the query template in Figure 39 is just a chain. If we replace the last edge with the conjunction of two differently labeled edges, as shown in Figure 45, then we have replaced some OR-parallelism with AND-parallelism. If

simple clause of the form:

$$q(X, Y) :- p(X, Y).$$

where p matches in the AG but q only matches the clause head. There is still only OR-parallelism in such a query but it requires $2^h - 1$ invocations in addition to sending $2^{h+1} - 1$ query messages. This increases the number of logical inferences needed to solve the query by 50% since a clause invocation is required for each pair of query messages.

Case 2: Simple Invocation with Multiple Clauses

After each invocation in Case 1, the clause body literal $p(X, Y)$ matches two AG edges. This is the source of OR-parallelism in Case 1. In this case, the invocation will be the source of OR-parallelism in the following way. Give each pair of AG edges two different labels as in Section 4 and let each query chain edge match two different clause heads:

$$q(X, Y) :- g(X, Y).$$

$$q(X, Y) :- d(X, Y).$$

where g and d match in the AG. Here the number of query messages remains the same as in Case 1 since the same number of AG edges must be matched but $2^{h+1} - 2$ invocations are required. This doubles the number of logical inferences needed to solve a query.

h_a	1	2	3	4	5
%	50.0	75.0	87.5	93.8	96.9
t_x	22.07	22.09	21.63	22.63	22.79

Table 3

Simulation Results under AND-parallelism

same annealing parameters as before were used where $|AG| = 2047$ and $|PE| = 256$. Note that $h_a = 1$ requires a query of 11 literals whereas $h_a = 5$ requires a query of 67 literals while the AG consists of 2046 literals (edges). Here the overall query times are essentially constant. The overall activity is also very similar as shown in Figure 46. This demonstrates that BPEM handles AND-parallelism just as well as OR-parallelism and by implication, the performance under AND-parallelism will scale-up as well. The average message length was again ≈ 8.5 characters for all queries.

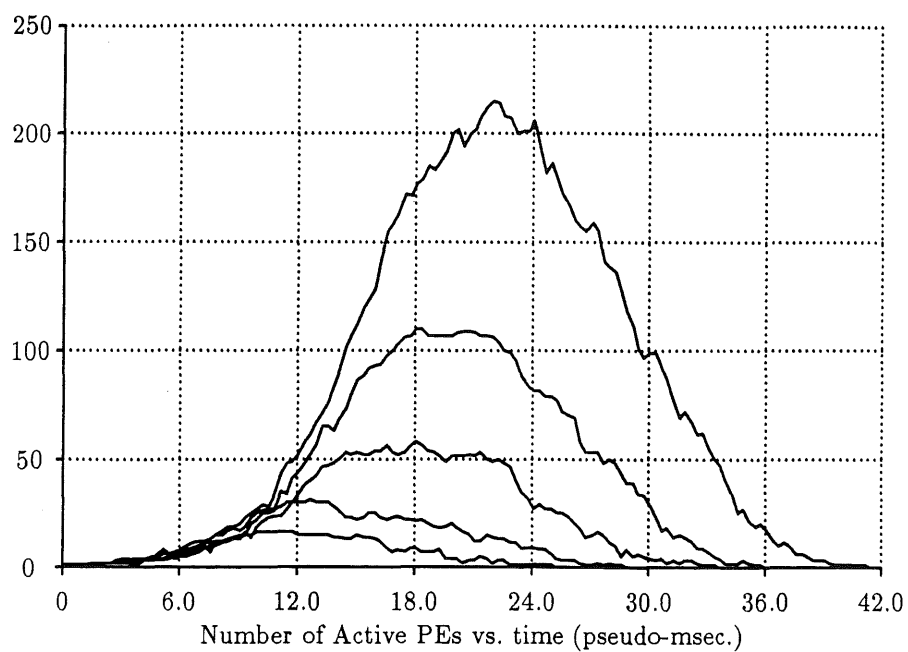
5. Clause Invocation

In this section, we present further twists on the original OR-parallelism problem to demonstrate the effect of clause invocation (clause head unification) on BPEM's performance. We simulated three cases. The first is simple invocation where matching each AG edge requires one invocation. The second is simple invocation where the OR-parallelism is derived from invoking multiple clauses rather than from matching multiple edges in the AG. The third is nested invocation where a chain of invocations is done.

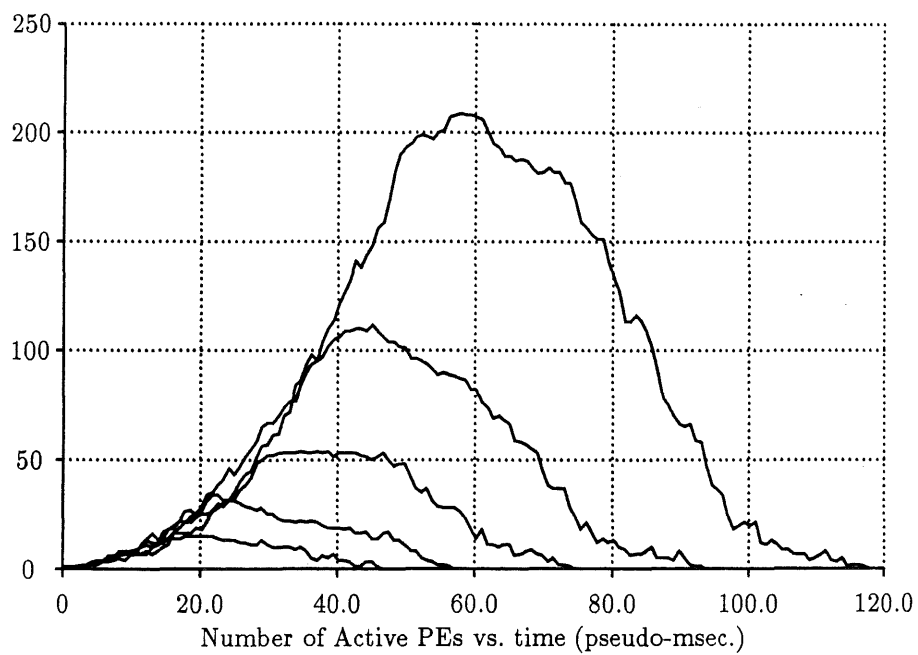
5.1. Problem Statements

Case 1: Simple Invocation

In the original problem, the query template was a chain that directly matched AG edges. Here, each edge in the chain will be a reference to the head of one very

**Figure 48**

Activity Curves under Simple Invocation with OR-parallelism

**Figure 49**

Activity Curves under Nested Invocation

h	6	7	8	9	10
Case 1					
t_x	21.79	28.19	30.63	33.23	40.04
t_x/h	3.63	4.03	3.83	3.69	4.00
k-lips	8.72	13.55	25.01	46.16	76.67
Case 2					
t_x	25.07	28.84	33.56	35.87	41.33
t_x/h	4.18	4.12	4.20	3.98	4.13
k-lips	10.09	17.65	30.42	57.01	99.03
Case 3					
t_x	46.01	56.68	74.31	93.39	117.67
t_x/h	7.67	8.10	9.23	10.38	11.77
k-lips	4.13	6.74	10.31	16.42	26.09
a.m.l.	61.9	73.2	85.6	98.2	109.7

Table 4
Simulation Results under Clause Invocation

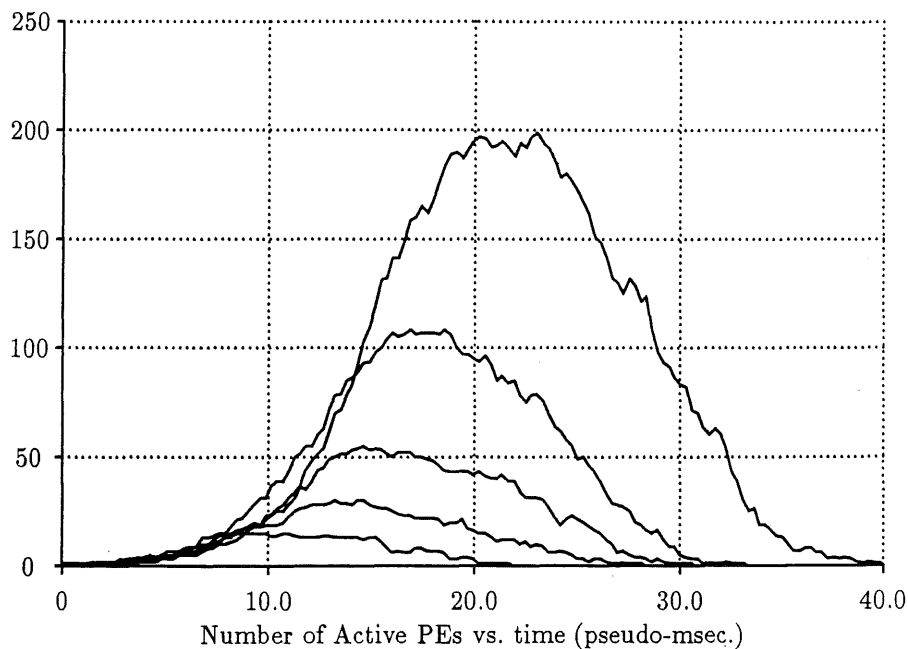


Figure 47

Activity Curves under Simple Invocation

Case 3: Nested Invocations

This case demonstrates a situation where the model's performance does not scale-up. When invocations are nested, as in recursion, each descendent must be sent information about all clauses that are being "worked on" such that messages become longer and require more processing. Here the query templates are still chains but will require an invocation to add the next edge in the chain as matching proceeds. Hence, for AG height h , the program would have the structure

```

:- q1( a, X ).
q1( X, Z ) :- p( X, Y ) [ q2( Y, Z ) ].
q2( X, Z ) :- p( X, Y ) [ q3( Y, Z ) ].
q3( X, Y ) :- p( X, Y ) [      ...      ].
.
.
.
qh( X, Y ) :- p( X, Y ).

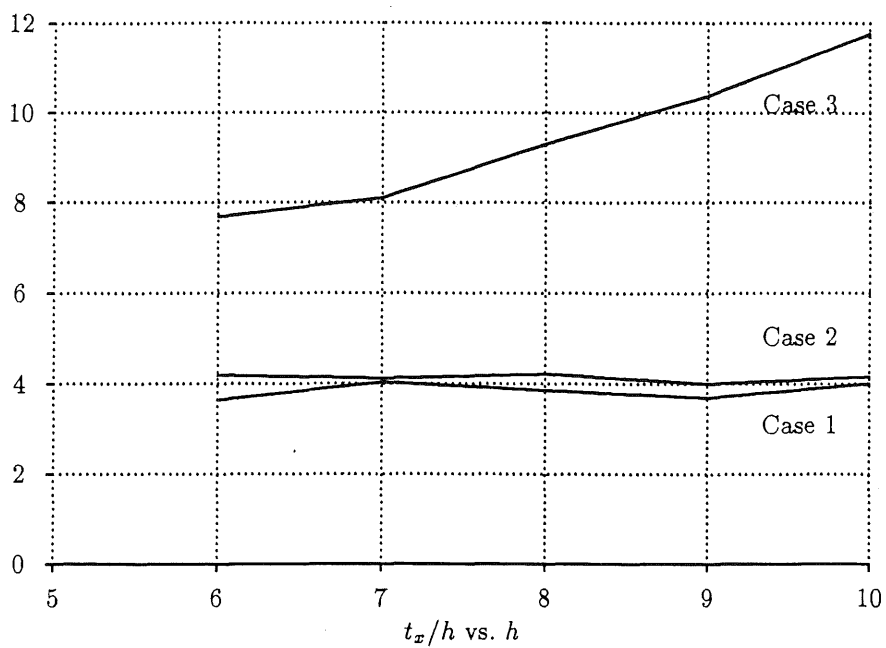
```

where the predicate p matches twice in the AG and the constant a is its root. There is still only OR-parallelism in this query (from the AG) and it requires $2^h - 1$ invocations and $2^{h+1} - 1$ query messages (same as Case 1).

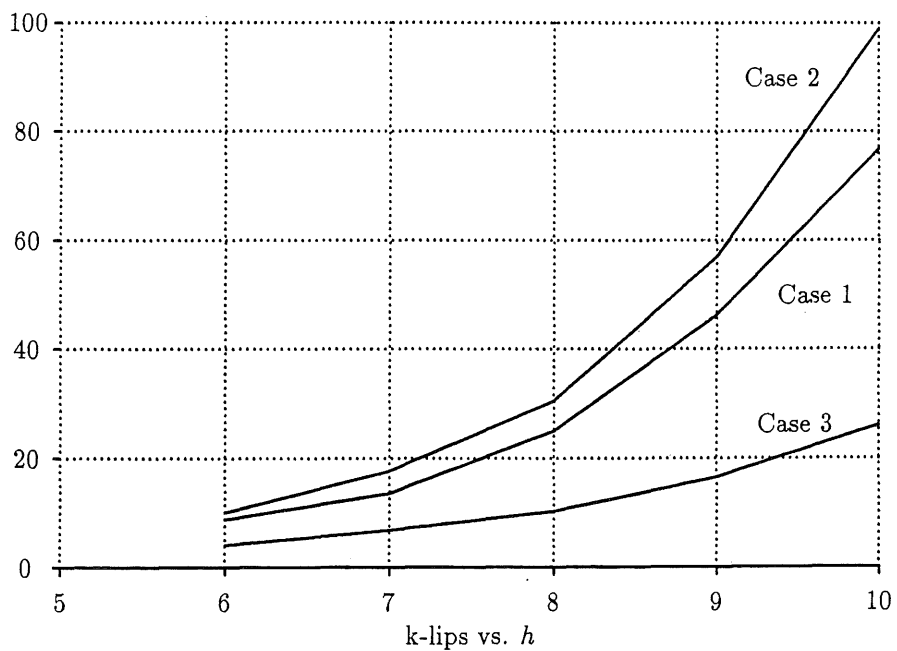
5.2. Results

In all cases, $|AG| = 8 \cdot |PE|$ was maintained as the average case of the preceding simulations and the same annealing parameters as in Section 3 were used. Table 4 summarizes the performance for all three cases. The activity curves are shown in Figures 47, 48 and 49, respectively. The t_x/h and k-lips curves are shown in Figures 50 and 51, respectively. The significance of these are discussed for each case.

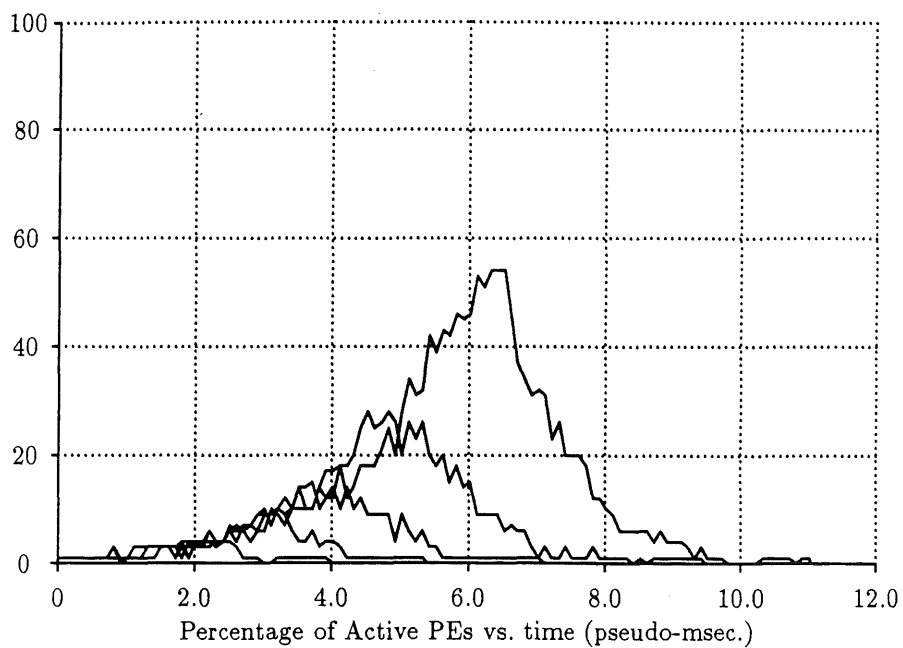
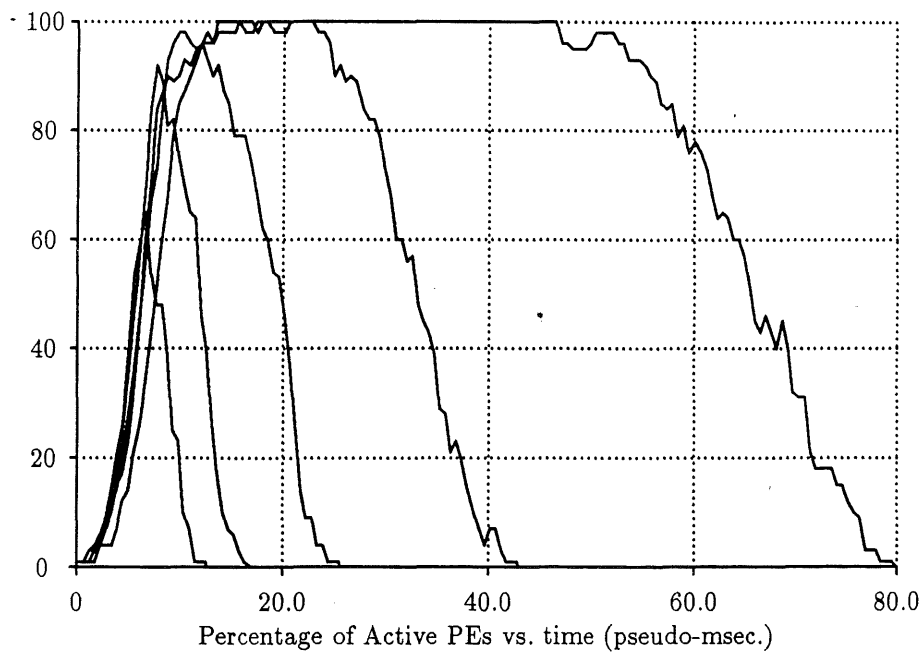
In Case 1, the overall query times have almost doubled when compared with the 8x case in Section 3. This is reflected in the average message length for all queries of ≈ 16.3 characters. The k-lips, however, have only slightly decreased

**Figure 50**

Relative Execution Times under Invocation for all Cases

**Figure 51**

K-lips under Invocation for all Cases

**Figure 52**Activity Curves for $2 \leq h \leq 6$ **Figure 53**Activity Curves for $7 \leq h \leq 11$

($\approx 17\%$ at $h = 10$) since more inferences are being done, i.e., clause invocations in addition to query messages are being produced. The t_x/h ratios have also increased but they are again essentially flat as seen in Figure 50. This suggests that while clause invocation is a more expensive operation, it is still linear such that the model's performance still scales-up.

In Case 2, the queries take slightly longer but since twice as many logical inferences are being done, the k-lips are higher. This implies that the second invocation is cheaper than the first. As a result, the k-lips curve is significantly higher than Case 1. The t_x/h curve is also flat and only slightly above that of Case 1.

In Case 3, the performance has suffered significantly as shown by the longer overall query times and also the lower k-lips that are attained. The performance is not scaling-up as well as shown by the increasing t_x/h ratio. This reflects the fundamental property of BPEM that information about each invocation must be passed to all descendants, thus increasing the amount of work that must be done after each one. This is reflected in the increasing average message lengths (a.m.l., in characters).

6. A Spectrum of Problem Sizes

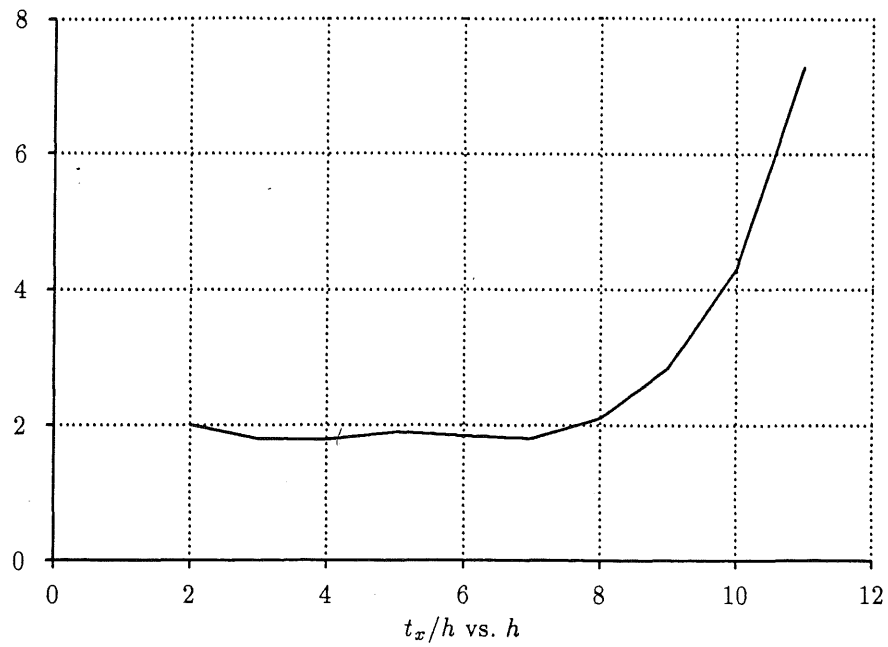
In this section, rather than showing how the model scales-up by maintaining a constant ratio between the problem and network sizes, we fix the network size and demonstrate the behavior of the model for a spectrum of problem sizes; from very small to very large compared to the network. These simulations show the performance when the network is hardly used at all up to when the network is saturated with work. Besides showing activity curves, we will show curves of the average and maximum queuing of messages over the entire network.

$ AG / PE $	h	t_x	t_x/h	k-lips
1/8	2	4.01	2.00	1.74
1/4	3	5.37	1.79	2.79
1/2	4	7.12	1.78	4.35
1	5	9.48	1.90	6.64
2	6	11.03	1.84	11.51
4	7	12.63	1.80	20.19
8	8	16.85	2.11	30.33
16	9	25.53	2.84	40.07
32	10	42.92	4.29	47.69
64	11	80.04	7.28	51.16

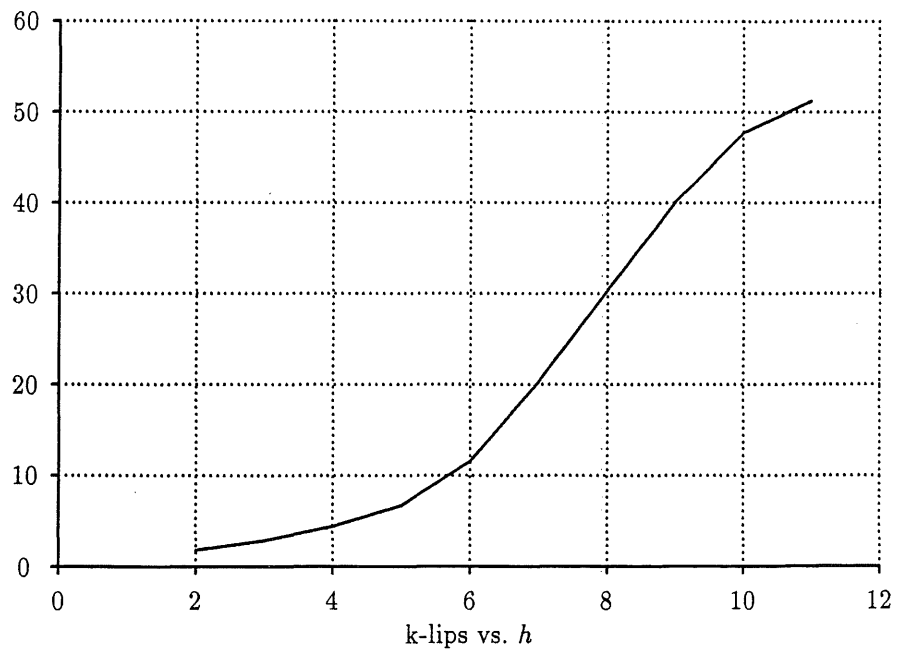
Table 5

Simulation Results for a Spectrum of Sizes

The network size was fixed at 64 PEs. The Binary Tree Search problem was used as in Section 3 along with the same annealing parameters. The problem size was varied from 1/8 to 64 times the network size which also varies h from 2 to 11 as indicated in Table 5. The activity curves are shown in Figures 52 and 53 where the vertical scale is given in percent from 0 to 100 since the number of PEs is fixed. Clearly, for $h = 10$ and $h = 11$, the network has become saturated. If we now look at the t_x/h ratios, as plotted in Figure 54, we see that the network is providing maximum parallelism until about $h = 8$, i.e., until the problem is about 8 times the network size. Above this point, the exponential size of the problem is reflected in exponential execution times since the simulated machine can utilize only a finite number of PEs. What this means in terms of k-lips, is that instead of a constantly rising rate (as when the network size is increasing), we see an s-curve, as shown in Figure 55, where the network is reaching an asymptote of maximum performance between 50 and 60 k-lips as the utilization approaches 100% over the course of the computation. This suggests that each PE is capable of ≈ 0.9 k-lips.

**Figure 54**

Relative Execution Times Approaching Saturation

**Figure 55**

K-lips Approaching Saturation

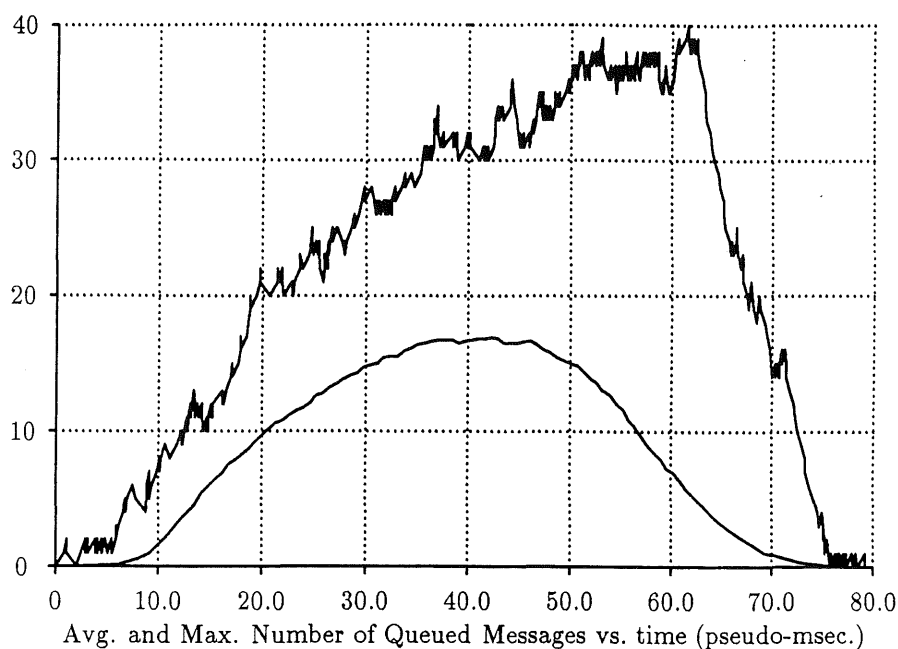
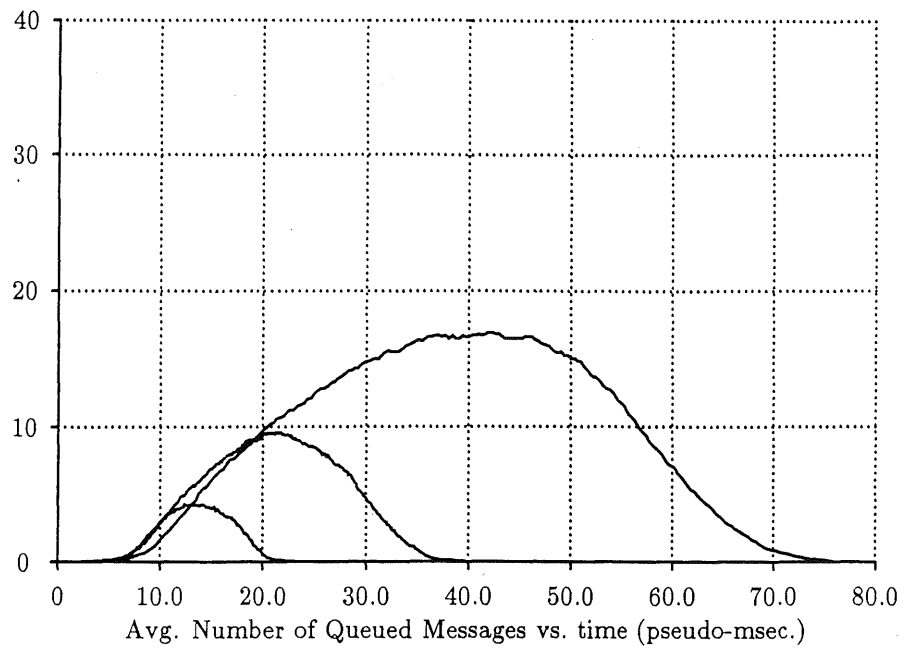


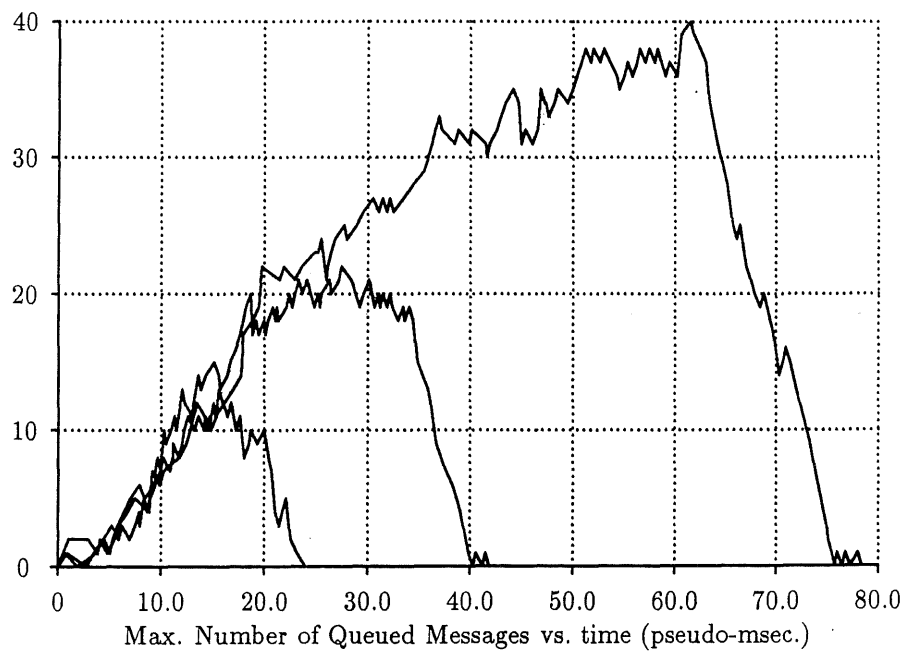
Figure 58

Average and Maximum Message Queuing for $h = 11$

With regards to queuing, Figures 56 and 57 show the average and maximum message queuing per PE, respectively, for $h = 9, 10$ and 11 . (Some smoothing has been done on Figures 57 and 58 to reduce the amount of data for plotting and typesetting.) Negligible queuing occurs for $h < 9$. Not surprisingly, the queuing increases the most when the network is saturating. If we look at the average and maximum queuing for $h = 11$, as shown in Figure 58, we see an interesting phenomenon. The maximum queuing reaches its peak *after* the average queuing does. At $t = 60.0$, the average PE has about 7 messages queued-up while at least one PE has almost 40 messages. This indicates hot-spots that are probably due to the reply messages that are generated late in the computation and converge on fewer PEs as they propagate back up the tree to the root. While this particular pattern of queuing is specific to this problem, it does demonstrate the need for some type of load balancing or, alternatively, *variable service* as discussed in Chapter 5.

**Figure 56**

Average Message Queuing for $h = 9, 10$ and 11

**Figure 57**

Maximum Message Queuing for $h = 9, 10$ and 11

$ AG / PE $	h	t_x	hop/msg
1/8	2	4.72	1.57
1/4	3	7.84	1.47
1/2	4	8.75	1.29
1	5	11.38	1.33
2	6	12.70	1.01
4	7	15.88	0.65
8	8	17.20	0.55
16	9	26.65	0.47
32	10	42.10	0.44
64	11	79.55	0.44

Table 6

Simulation Results for a Spectrum with Independent Routers

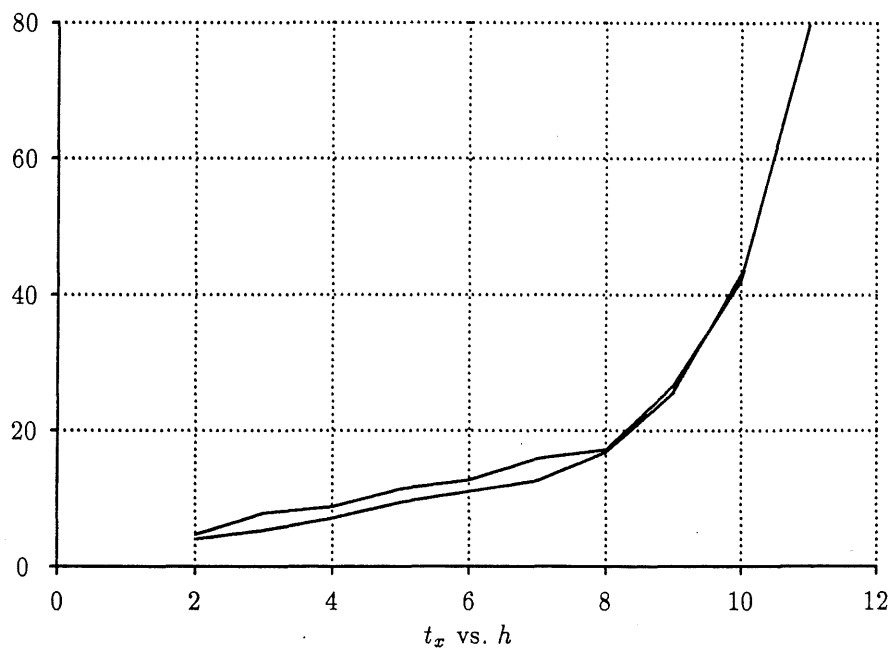


Figure 59

Execution Times with and without Independent Routers

for the larger problems. This is shown graphically in Figure 59. We can see from the average hops/msg that there is not very much routing that must be done, hence, the similar performance. These results indicate that simulated annealing is working quite well in this situation; again, especially for the larger problems. The

The number of queued messages does not pose any problems for BPEM, per se; it is only indicative of a saturated machine or a hot-spot. Since all physical machines are finite, there is always some problem that is large enough to saturate it. If a problem has an inherent bottleneck, then a hot-spot is likely. Larger problems can be accommodated by larger machines. Hot spots can be alleviated by the variable service technique mentioned above. In any physical implementation, however, the length of the queues in which messages must reside is finite. In this case, the issues of queue overflow and deadlock would have to be addressed as discussed at the end of Chapter 5.

7. Independent Routers

In Chapter 5, the use of independent routers was described to improve system performance. The most improvement will be realized, however, only when the average number of hops per message is greater than one. If this is not the case, then an independent router is only saving the associated PE the work of routing its own messages and not the larger routing task of relaying those messages that are simply being forwarded.

This section demonstrates this phenomenon by repeating the spectrum of simulations from the previous section but using independent routers. This was simulated by having the originating router compute the number of hops a message requires, adjusting its time-stamp accordingly, and sending the message directly to its destination. While this simulates the propagation time of a message, it does not simulate the contention a message would encounter at each router. This does not, however, affect the results presented here.

Table 6 presents the execution times, t_x , and the average number of hops per message, hop/msg, for the same queries as in the previous section. One can note immediately that the execution times follow closely their previous values; especially

not expected to be a serious drawback, however, since semantic network queries are not expected to be as deeply nested as general logic programs.

These statements of BPEMs performance are, of course, tied directly to this software implementation. A more efficient implementation, especially concerning the management of storage, would certainly yield better results. Hardware support would also greatly speed-up processing. Besides off-loading message routing (which is currently done in software), each PE could enjoy multiple message-servers. Message-servers could even be shared between PEs such that service could be skewed to busier PEs in order to alleviate hot-spots.

fact that the average hops/msg decreases from 1.57 to 0.44 as the problem size increases demonstrates that simulated annealing with the distance-variance metric is able to evenly distribute the AG while minimizing communication when there is more AG to distribute relative to the size of the network.

Of course, these results depend on the relative cost of transmitting a message and routing a message and also of enqueueing and dequeuing messages. Since it was not the original intent of the simulator to model this level of hardware, more properly designed simulations should be done that could include such factors as router contention.

8. Conclusions

Basic simulation results of the BPEM have been presented. We have shown that the performance of the model scales-up under OR-parallelism and AND-parallelism. While it is not surprising that they are both linear time operations, it is surprising that they take essentially the same amount of time. If we assume that every simulator instruction takes 1 μ sec., then 256 PEs can deliver on the order of 10^5 k-lips regardless of the mix of OR and AND-parallelism. This is quite realistic given that the simulations with saturated PEs suggest a peak performance of ≈ 0.9 k-lips/PE (for this implementation) which scales to ≈ 230 k-lips for 256 PEs.

Performance is sensitive to clause invocation which is a more expensive operation. While the performance of 256 PEs dropped to ≈ 76 k-lips for single invocations, it still scaled-up as the problem and network sizes were increased. Doing multiple invocations for each literal is a little more efficient in that the performance only dropped to ≈ 99 k-lips. We also demonstrated the effect that nested invocations have on performance. As nesting deepens, each query message becomes longer and requires more processing. As a result, the performance no longer scales-up, i.e., as the problem/network sizes increase, the performance degrades. This is

little time must be spent relaying messages. The quality of the map also depends, however, on the structure of the logical architecture. It is possible that for some graphs, no good map exists, thus requiring a significant number of hops per message. Regardless of how low the average number of hops is, however, time must still be spent encoding and decoding messages and this source of overhead may be reflected in the relatively low performance of an individual PE. This overhead, however, is proportional to the size of the problem in the absence of nested clause invocation, i.e., the overhead per message is constant. Thus, the asynchronous message-passing of BPEM can successfully scale-up to larger machines and larger problems to allow massive parallelism. It imposes no limits on the size of problem to which it can be applied due to any kind of centralized controller or memory.

The granularity of BPEM is generally a step of resolution but this can vary for several reasons. The first is that a message may cause one or more clause invocations, each of which is a step of resolution that may or may not directly generate any new messages. Such a message represents a much larger grain. Second, simulated annealing may allocate neighboring AG vertices to the same PE such that several steps of resolution may be done without physically sending any messages between PEs. This also represents a much larger grain. Hence, in a typical query, there is much parallelism that is available but the granularity is not as fine as it could be thus lowering the communication overhead.

BPEM derives specific benefits from using logic as a programming paradigm. One very important benefit is that multiple top-level queries can be executed in an OR-parallel fashion since they are logically independent. Logic also allows one to concentrate on the logical solution without having to pay as much attention to control issues. This is especially evident when compared with other semantic network systems that use a set of system-specific primitive operations as described in the next section on related work. Such primitives have little to do with the

CHAPTER 7

Conclusions and Future Research Directions

1. Contribution

The contribution of this work is the development of a distributed, massively-parallel system for semantic networks and knowledge bases that is built on a subset of first-order predicate logic. The reliance on logic gives the model an easily-understood programming paradigm and a well-defined semantics of execution. The parallelism is extracted from a graphical representation of logic based on the binarization of predicates; hence, the name Binary Predicate Execution Model. Execution is supported by asynchronously passing messages between program constants along edges defined by the program facts. Under most conditions, the model's performance scales-up for larger problems and machines, demonstrating that it does not introduce any artificial locality or computation bottlenecks.

2. Discussion

The overall performance of this model is favorable given that we have examined a relatively simple software implementation. This suggests a performance of ≈ 0.9 k-lips/PE or on the order of 10^5 k-lips for a 256 PE n -cube. Of course, this figure includes the communication overhead which can vary according to the size of the problem and the quality of the map found by simulated annealing. Even so, the performance per PE is similar to that of the first sequential implementations of logic languages.

We have shown that simulated annealing can produce very adequate maps such that the average number of hops per message is < 1.0 meaning that very

The Semantic Network Array Processor (SNAP) is similar but a little more complicated than NETL. It consists of a grid of PEs with local communication in addition to a global bus managed by a synchronous controller [MT85]. The nodes of a semantic network are mapped one-to-one on to the PEs and the global controller broadcasts a sequence of primitive instructions that are carried out in parallel by the PEs. Here, each PE has a list of logical neighbors in the semantic network and a set of general purpose flags. The broadcast instructions involve searching the list of neighbors, testing the flags and doing logical operations on them. Messages can also be sent on the local grid to neighboring PEs. These messages, however, are more than just a marker in that they can set a specific flag in a neighbor and they are queued rather than immediately OR-ed. More complex application functions that can be built out of the primitive instructions including pattern matching, production systems and inferencing. Image understanding can also be done given that the elements of a scene are represented as a graph [DM87].

The simple nature of marker-passing systems ensures that processing is done quickly. For set operations like intersection, marker-passing is ideal. Marker-passing is insufficient, however, for any reasoning involving probabilities or confidence levels. Another drawback is that the set of primitive instructions have a semantic gap between them and the application level. To ease the burden of hand-coding higher-level operations, a compiler would be very helpful in generating the required sequence of instructions. In the case of both NETL and SNAP, the global bus limits the size of the machine and the size of feasible problems. The size of feasible problems will also be limited if each PE can handle only one node. Even with one-to-one mapping, the problem of mapping nodes on to PEs must still be adequately solved to minimize communication costs.

application and, hence, require the development of an algorithm to solve the problem at hand. Control becomes an issue again, however, with the introduction of non-logical extensions. These do allow the user to specify control and also to blur the distinction between forward and backward chaining by using variable attributes as memory. They also allow "incremental" queries; partial queries may be processed independently in time but still have an effect on later ones.

3. Related Work

BPEM is related to the concept of *connectionism* in AI in that some part of the problem is viewed as a massively-parallel *active graph* of relatively simple elements that process more complex problems. The architectures that support connectionism can be classified according to how they pass information and the programming paradigm that is used to organize the computation: *marker-passing*, *value-passing* and *message-passing* [FHS83]. This provides a useful framework for BPEM's comparison with other parallel systems for semantic networks and knowledge bases.

3.1. Marker-Passing

Marker-passing systems are the simplest since PEs exchange only single-bit marker messages along the network links. This is typified by NETL where PEs are the nodes of a semantic network and may store a few markers. Simple operations may be done on these markers and PEs may be associatively polled for a given marker pattern to accomplish search and deduction within the network [FAHL79]. While many markers may be flowing between PEs, there is no contention since it is assumed that dedicated lines between PEs exist and that multiple markers arriving at the same node are simply OR-ed together. The instructions for doing these operations, however, are organized and broadcast over the entire network by a synchronous external controller.

While both the Connection Machine and Shastri's connectionist encoding "circuits" can be termed value-passing, the contrast between their method of communication and capabilities is very interesting. Shastri's system can only do inheritance and categorization; no other operations were discussed. It does this, however, just by asynchronously passing simple value messages. The Connection Machine, on the other hand, can do more complex operations like inferential reasoning but uses synchronous broadcast of primitive instructions to all PEs. A program for a connectionist encoding must be "hard-wired" into the "circuit"; that is to say, it really doesn't have a programming paradigm. A program for the Connection Machine is a sequence of primitive instructions sent over a separate channel similar to NETL and SNAP. Thus, it should not be surprising that an encoding may be faster for a particular problem but the Connection Machine is more general.

3.3. Message-Passing

Message-passing systems are the most powerful and the most complex since messages of arbitrary complexity are allowed. BPEM clearly falls in this category. While the simpler cases can be viewed as possible models of information processing in the brain itself, message-passing is certainly too complex for this purpose. For the processing of semantic networks, however, message-passing allows the specification of more complex operations to be carried on the messages themselves.

One example is IX. This is a pyramid-shaped, packet-switched network with PEs that have associative memory which supports the language IXL [HHKF86]. IX the machine appears to be an asynchronous, message-passing system whose shape is suggested by a logic program's execution tree. IXL the language specifically supports semantic networks by creating nodes, links and inference rules. Different kinds of links are possible: *isa*, *instance_of*, *a_kind_of* and source/destinations links for relations. A distinction is made between properties and assertions. An assertion

3.2. Value-Passing

Value-passing systems remove some limitations of marker-passing by passing bit vectors that can represent integers or floating-point numbers. One example of this is Thistle which uses eight-bit values [FHS83]. A Thistle PE can do simple arithmetic operations on these values and can handle multiple incoming messages by finding the *min* or *max* of their values.

Another example is the Connection Machine [HILL84, TR88]. Messages in the Connection Machine have a type and a value that can be a number or a pointer. Each PE has a group of set-registers which contain sets of nodes in a semantic network. Each PE also has a group of function-registers which contain "functions", i.e., the network links which "map" nodes to nodes or nodes to values. Different groups of operations can be done on these registers. These groups include set operations, propagation of values for property inheritance, "function" manipulation and arithmetic. While the Connection Machine does have a synchronous, broadcast controller, it also has nearest-neighbor grid communications.

In contrast to the Connection Machine, Shastri reports an asynchronous, value-passing system with no central controller that uses *evidential reasoning* to find most likely conclusions [SHAS87]. Shastri calls this a *connectionist encoding* of semantic networks and can handle inheritance, categorization (the dual of inheritance), exceptions and conflicting information by using evidential reasoning. A network is built of five different units: *concept*, *property*, *binder*, *relay* and *enable* nodes which exchange real-valued potentials in the range $[0,1]$. A property P with a value V is said to support a concept A with a strength computed by a binder node. Enable nodes determine the type of query (inheritance or categorization) while relay nodes determine the directionality of computation. A query is specified by setting certain external inputs to 1.0 and after a number of time steps related to the size of the network, the solution is found.

final solutions. (This may be related to another model for doing set operations between pairs of semantic networks [SAP83]. Here, an input "block" injects a textual representation for one network into all PEs connected as the other network and all partial solutions are reported to a single output "block".) Such "common channels" have the possibility of becoming bottlenecks and limiting performance.

A major difference between the message-passing models and the others like NETL and the Connection Machine is that since instructions are not *broadcast* over the entire network, they must be somehow included in the messages themselves. Hence, they communicate asynchronously like the connectionist encoding yet the added message complexity allows it to support more complex formalisms such as resolution in first-order logic. This eliminates the need for broadcast communication. Thus, we have a *distributed* inference engine which is sufficient to support such formalisms such as semantic networks and knowledge bases. The fact that there is no broadcast communication and that the system is distributed means that there is no size limit imposed by the model.

Another major difference between BPEM and all of these models is the use of *logic* as the supporting formalism. This provides a clear, well-defined programming paradigm that is integrated into the execution model. One does not encounter the same semantic gap between the application problem and a set of primitive instructions as in most of the systems discussed above. Whether logic is sufficient to represent *knowledge*, however, is another topic of discussion [ISRAEL83]. Regardless of arguments pro and con, one cannot deny that logic is capable of many things including the organization of massively-parallel computation.

is a relation between sources and destinations themselves. A property is a relation between *instances* of sources and destinations. IXL is a logic-like language which may be influenced by the fact that it is implemented in Prolog. All links and inference rules are written as predicates and clauses. While IXL has a basis in logic as a programming paradigm, the restriction to higher-level semantic network concepts has the effect of prohibiting some of what logic can do. The authors state, for example, that IXL is limited when dealing with sets, quantification or the "grouping" of knowledge.

Another example is proposed by Sapaty [SAP86]. This model is perhaps the closest in spirit to BPEM yet its programming paradigm is completely different. Here, a semantic network is also envisioned as a graph of active PEs that asynchronously pass messages. Queries are written, however, in a *wave* language. This is a family of languages and the most basic is WAVE-0 which expresses primarily navigational information. Each query message consists of a concatenated sequence of primitive operations. When a message is received, the head operation is performed and the tail can be replicated and sent to other PEs according to edge name and direction. The spreading of messages is referred to as a *wavefront* and the contents of a message as a *search-image*. In contrast to BPEM, wavefronts can be synchronized by using *hierarchical colorings*, i.e., path names of unbounded length.

Clearly, WAVE-0 is not a logic language and queries must be built from primitive operations that are not related to the content of the semantic network. Hence, like SNAP, there exists a semantic gap that must be bridged. Ironically, a Prolog-like example is given that matches binary predicates. The processing, however, does not follow the structure of the goal list but rather the structure of primitive operations. All nodes try to match to all variables and all partial matches are output to a "common channel" that finds the union of these to produce the

to *adjust* granularity would be advantageous but appears difficult since granularity depends in part on the mapping which can have a different effect on the granularity of different queries.

Syntactic Entities vs. Semantic Implementation. The granularity of BPEM and other such systems is also determined by the relationship between the syntactic entities of a program and the implementation of their semantics. BPEM, for example, associates active processing with constants and derives parallelism from unifying facts. Other systems, as described in Chapter 3, associate processing with literals and clauses and derive parallelism from unification with either facts or clauses. The effect that each approach has on the granularity is interesting not only for implementations but also from a basic science standpoint.

More General Theorem Proving. Many graph-oriented methods for proving theorems in one form or another have appeared in the literature and one cannot help but wonder about the possible parallelism and required overhead of such systems. Gallier and Raatz describe HORNLOG [GR85, GR87] which uses general Horn clauses and is based on graph rewriting rules for connection graphs [CS79, Kow75]. Since general Horn clauses are used, assertions and indefinite answers are possible. Even though the authors mention definite parallel possibilities, no work has been done on it as of this writing [RAATZ88]. Bibel describes a method of theorem proving by finding mating between matrices which is similar to work by Andrews [BIBEL83, ANDR81]. Here general formulae are represented in matrix form and solutions, called matings, are found by building paths through connections of negated and unnegated literals. This method also has tremendous opportunities for parallelism. Of course, this is no longer using resolution. In fact, there are many non-resolution strategies for theorem proving [BLED77] which could very well offer many different highly-parallel methods for solving "intelligent" problems.

4. Future Work

There are many potential areas of further work related to BPEM. Some of these concern just implementation issues while others have a more general concern of parallelism and logic.

WAM-like Storage Model and Instruction Set. The Warren Abstract Machine (WAM) has provided a coherent model for storage and instructions for sequential execution [WARR83]. Since it is expressed at an abstract level, any component could be implemented in software, firmware or hardware. BPEM could also benefit from a similar unified approach to storage and the processing of each different message type.

Overhead. The simulations of Chapter 6 provide a good idea of the potential overall performance but do not quantify the cost of each type of overhead. Such an analysis could identify overhead costs that need to be reduced such as the encoding and decoding of messages. Of course, any such analysis would be implementation-specific. Another implementation strategy, such as derived from an abstract machine, could change the cost of various overheads.

Load Balancing. The saturation simulations demonstrate the general need for some form of load balancing. While not all problems will have unavoidable hot-spots, some will. Given the overhead of migrating logical vertices, the variable service hybrid architecture described in Chapter 5 deserves further investigation and simulation. This provides flexibility in the allocation of service with a lower overhead.

Granularity. This is related to both overhead and load balancing in that overhead is associated with each grain and load balancing depends on the number and size of grains relative to the size of the problem and machine. Hence, the distribution of grain sizes in BPEM needs to be more accurately characterized in order to ascertain whether the average grain is too large or too small. Being able

- [BF81] BARR, A., FEIGENBAUM, E.A. *The Handbook of Artificial Intelligence, Vol. 1*, HeurisTech Press, Stanford, California, 1981.
- [BH83] BROOMELL, G., HEATH, J.R. Classification Categories and Historical Development of Circuit Switching Topologies. *Computing Surveys V15*, n2 (1983), 95-133.
- [BIBEL83] BIBEL, W. Matings in Matrices. *Comm. Assoc. Computing Machinery V26*, n11 (Nov., 1983), 844-852.
- [BIC84] BIC, L. A Data-Driven Model for Parallel Interpretation of Logic Programs. *International Conference on Fifth Generation Computer Systems*, Tokyo, Japan (1984).
- [BL87] BIC, L., LEE, C. A Data-Driven Model for a Subset of Logic Programming. *Trans. on Prog. Lang. and Sys. V9*, n4 (October, 1987), 618-645.
- [BLED77] BLEDSOE, W.W. Non-resolution Theorem Proving. *Artificial Intelligence V9* (1977), 1-35.
- [BOK81] BOKHARI, S.H. On the Mapping Problem. *IEEE Trans. on Computers VC-30*, n3 (March, 1981), 550-557.
- [BORG84] BORWARDT, P. Parallel Prolog using Stack Segments on Shared -memory Multiprocessors. *1984 Logic Programming Symposium*, 2-11.
- [BRACH79] BRACHMAN, R.J. On the Epistemological Status of Semantic Networks. In *Associative Networks*, N.V. Findler, ed., 1979, pp. 3-50.
- [BRACH83] BRACHMAN, R.J. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *Computer V16*, n10 (October, 1983), 30-36.
- [BRUY82] BRUYNOOGHE, M. The Memory Management of Prolog Implementations. In *Logic Programming*, Clark & Tärnlund, eds., 1982, pp. 83-98.
- [BS81] BURTON, F.W., SLEEP, M.R. Executing Functional Programs on a Virtual Tree of Processors. *1981 Conference on Functional Programming Languages and Computer Architecture*, 187-194.
- [CF58] CURRY, H.B., FEYS, R. *Combinatory Logic*, North-Holland, 1958.

REFERENCES

- [AGAR83] AGARWAL, K.K. Solving Problems in Robotics with Semantic Networks. *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-5*, n2 (March, 1983), 213-217.
- [AHK87] AKERS, S.B., HAREL, D., KRISHNAMURTHY, B. The Star Graph: An Attractive Alternative to the n -Cube. *1987 Int'l. Conf. Parallel Processing*, 393-400.
- [AHS85] ACKLEY, D.H., HINTON, G.E., SEJNOWSKI, T.J. A Learning Algorithm for Boltzmann Machines. *Cognitive Science V9* (1985), 147-169.
- [AK86] AKERS, S.B, KRISHNAMURTHY, B. A Group Theoretic Model for Symmetric Interconnection Networks. *1986 Int'l. Conf. Parallel Processing*, 216-223.
- [AL82] ARDEN, B.W., LEE H. A Regular Network for Multicomputer Systems. *IEEE Trans. Computers VC-30* (Jan., 1982), 60-69.
- [AMD67] AMDAHL, G.M. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conf. Proc. V30* (1967), 483-485.
- [ANDR81] ANDREWS, P.B. Theorem Proving via General Matings. *JACM V28*, n2 (April, 1981), 193-214.
- [AG82] ARVIND, GOSTELOW, K.P. The U-Interpreter. *Computer V15*, n2 (February, 1982), 42-49.
- [AGP78] ARVIND, GOSTELOW, K.P., PLOUFE, W. An Asynchronous Programming Language and Computing Machine. In *Advances in Computing Science and Technology*, Ray Yeh, ed., 1978.
- [AS83] ANDREWS, G.R., SCHNEIDER, F.B. Concepts and Notations for Concurrent Programming. *Computing Surveys V15*, n1 (March, 1983), 3-43.
- [BAC78] BACKUS, J. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *CACM V21*, n8 (August, 1978), 613-641.
- [BB87] BERGER, M.J., BOKHARI, S.H. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans. on Computers VC-36*, n5 (May, 1987), 570-580.

- [COHKJ87] COHEN, P.R., KJELDSSEN, R. Information Retrieval by Constrained Spreading Activation in Semantic Networks. *Information Processing and Management V23*, n4 (1987), 255-268.
- [COM83] *Computer*, Special Issue on Knowledge Representation V16, n10 (Feb., 1983).
- [CON83] CONERY, J. The AND/OR Process Model for Parallel Interpretation of Logic Programs. *Tech. Rep. 204*, (Ph.D. Thesis) (June, 1983), Department of Information and Computer Science, University of California, Irvine.
- [CON87] Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. *Preprint* (1987).
- [COOK83] COOK, S.A. An Overview of Computational Complexity. 1982 ACM Turing Award Lecture. *CACM V26*, n6 (1983), 401-408.
- [CRAM85] CRAMMOND, J. A Comparative Study of Unification Algorithms for OR-parallel Execution of Logic Languages. *IEEE Trans. on Computers VC-34*, n10 (Oct., 1985), 911-917.
- [CS79] CHANG, C.L., SLAGLE, J.R. Using Rewriting Rule for Connection Graphs to Prove Theorems. *Artificial Intelligence V12* (1979), 159-180.
- [CVET86] CVETANOVIĆ, Ž. Performance Analysis of Multiple-Processor Systems. Department of Electrical and Computer Engineering, University of Massachusetts (1986).
- [DEN80] DENNIS, J.B. Data Flow Supercomputers. *Computer* (November, 1980), 48-56.
- [DIJK65] DIJKSTRA, E.W. Cooperating Sequential Processes. In *Programming Languages*, F. Genuy, ed., 1968.
- [DIJK71] DIJKSTRA, E.W. Hierarchical Ordering of Sequential Processes. *Acta Informatica V1* (1971), 115-138.
- [DIJK75] DIJKSTRA, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *CACM V18*, n8 (1975), 453-457.
- [DK79] DELIYANNI, A., KOWALSKI, R. Logic and Semantic Networks. *Communications of the ACM V22*, n3 (1979), 184-192.

- [CG81] CLARK, K.L., GREGORY, S. A Relational Language for Parallel Programming. *1981 Conference on Functional Programming Languages and Computer Architecture*, 171-178.
- [CG86] CLARK, K.L., GREGORY, S. PARLOG: Parallel Programming In Logic. *Trans. on Prog. Lang. and Sys. V8*, n1 (Jan., 1986), 1-49.
- [CH74] CAMPBELL, R.H., HABERMANN, A.N. The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science V16* (1974), 89-102, Springer-Verlag, New York.
- [CH84] CIEPIELEWSKI, A., HARIDI, S. Control of Activities in the Or-parallel Token Machine. *1984 Symp. on Logic Programming*, 49-57.
- [CHP71] COURTOIS, P.J., HEYMANS, F., PARNAS, D.L. Concurrent Control with 'readers' and 'writers'. *Communication of the ACM V14*, n10 (Oct, 1971), 667-668.
- [CI84] CHU, Y., ITANO, K. Organization of a Parallel Prolog Machine. *1984 International Workshop on High-Level Computer Architectures*, 4.18-4.30.
- [CIEP84] CIEPIELEWSKI, A. Towards a Computer Architecture for OR-Parallel Execution of Logic Programs. *Tech. Rep. TRITA-CS-8401* (May, 1984), Department of Computer Systems, The Royal Institute of Technology, Stockholm, Sweden.
- [CK81] CONERY, J.S., KIBLER, D.F. Parallel Interpretation of Logic Programs. *1981 Conference on Functional Programming Languages and Computer Architecture*, 163-170.
- [CL73] CHANG AND LEE *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, New York, 1973.
- [CM81] CLOCKSIN, W.F., MELLISH, C.S. *Programming in Prolog*, Springer-Verlag, 1981.
- [CM82] CHEEMALAVAGU, S., MALEK, M. Analysis and Simulation of Banyan Interconnection Networks with 2×2 , 4×4 , and 8×8 Switching Elements. *1982 IEEE Real-Time Systems Symposium*, 83-89.
- [CM87] CERCONE, N., MCCALLA, G., EDS. *The Knowledge Frontier: Essays in the Representation of Knowledge*, Springer-Verlag, 1987.

- [GJ79] GAREY, M.R., JOHNSON, D.S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman & Co., San Francisco, 1979.
- [GKS87] GRANSKI, M., KOREN, I., SILBERMAN, G.M. The Effect of Operation Scheduling on the Performance of a Data Flow Computer. *IEEE Trans. on Computers VC-36*, n9 (Sept., 1987), 1019-1029.
- [GP85] GAJSKI, D.D., PEIR, J.-K. Essential Issues in Multiprocessor Systems. *Computer* (June, 1985), 9-27.
- [GR85] GALLIER, J.H., RAATZ, S. Logic Programming and Graph Rewriting. *Symposium on Logic Programming* (1985), 208-219.
- [GR87] GALLIER, J.H., RAATZ, S. Hornlog: A Graph-Based Interpreter for General Horn Clauses. *J. Logic Programming V4* (1987), 119-155.
- [GREG86] GREGORY, S. Parallel Logic Programming: The State of the Art. A report for European Computer Research Center.
- [GS81] GOODMAN, J.R., SEQUIN, C.H. "Hypertree: A Multiprocessor Interconnection Topology". *IEEE Trans. Computers VC-30*, n12 (1981), 923-933.
- [Gus88] GUSTAFSON, J.L. Reevaluating Amdahl's Law. *CACM V31*, n5 (May, 1988), 532-533.
- [HALIM86] HALIM, Z. A Data-Driven Machine for OR-parallel Evaluation of Logic Programs. *New Generation Computing V4* (1986), 5-33.
- [HEN75] HENDRIX, G. Expanding the Utility of Semantic Networks through Partitioning. *IJCAI* (1975), 115-121.
- [HEN79] HENDRIX, G. Encoding Knowledge in Partitioned Networks. In *Associative Networks*, N.V. Findler, ed., Academic Press, 1979, pp. 51-92.
- [HG85] HUDAK, P., GOLDBERG, B. Serial Combinators: "Optimal" Grains of Parallelism. *Conf. on Functional Programming Languages and Computer Architecture* (Sept., 1985).
- [HG87] HWANG, K., GHOSH, J. Hypernet Architectures for Parallel Processing. *1987 Int'l. Conf. on Parallel Processing* (August, 1987), 810-819.

- [DKM84] DWORK, C., KANELAKIS, P.C., MITCHELL, J.C. On the Sequential Nature of Unification. *J. Logic Programming V1* (1984), 35-50.
- [DM87] DIXIT, V., MOLDOVAN, D.I. Semantic Network Array Processor and Its Application to Image Understanding. *IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-9*, n1 (Jan., 1987), 153-160.
- [DvH66] DENNIS, J.B., VAN HORN, E.C. Programming Semantics for Multi-programmed Computations. *CACM V9*, n3 (1966), 143-155.
- [ECR84] ERCEGOVAC, M.D., CHAN, P.K., RAVI, T.M. A Data Flow Multimicroprocessor Architecture for High-Speed Simulation of Continuous Systems. *1984 Int'l Workshop on High-Level Computer Architecture*, 2.9-2.17.
- [EMK076] VAN EMDEN, M.H., KOWALSKI, R.A. The Semantics of Predicate Logic as a Programming Language. *JACM V23*, n4 (Oct., 1976).
- [FAHL79] FAHLMAN, S.E. *NETL: A System for Representing and Using Real-World Knowledge*, MIT Press, 1979.
- [FHS83] FAHLMAN, S.E., HINTON, G.E., SEJNOWSKI, T.J. Massively Parallel Architectures for AI: NETL, Thistle and Boltzmann Machines. *AAAI* (1983).
- [FIHE77] FIKES, R.E., HENDRIX, G.G. A Network-Based Knowledge Representation and its Natural Deduction System. *IJCAI* (1977).
- [FIN79] FINDLER, N.V. *Associative Networks*, Academic Press, 1979.
- [FISHER81] FISHER, J.A. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Trans. on Computers VC-30*, n7 (July, 1981), 478-490.
- [FNM82] FURUKAWA, K., NITTA, K., MATSUMOTO, Y. Prolog Interpreter Based on Concurrent Programming. *First Int'l Logic Programming Conf.* (1982), 38-44.
- [FRANK87] FRANK, M. Meeting, Dept. of Information and Computer Science, University of California, Irvine (August, 1987).
- [GAMI78] GALLAIRE, H., MINKER, J., EDS. *Logic and Data Bases*, Plenum, N.Y., 1978.

- [KLT84] KELLER, R.M., LIN, F.C.H., TANAKA, J. Rediflow Multiprocessing. *IEEE Comcon '84* (Feb., 1984), 410-417.
- [KM85] KASIF, S., MINKER, J. The Intelligent Channel: A Scheme for Result Sharing in Logic Programs. *IJCAI* (1985), 29-31.
- [Kow75] KOWALSKI, R.A. A Proof Procedure Using Connection Graphs. *JACM* V22, n4 (Oct., 1975), 572-595.
- [Kow79] KOWALSKI, R. *Logic for Problem Solving*, Elsevier North Holland, New York, 1979.
- [LA87] LEE, S.-Y., AGGARWAL, J.K. A Mapping Strategy for Parallel Processing. *IEEE Trans. on Computers* VC-36, n4 (April, 1987), 433-442.
- [LEE85] LEE, C.A. Cost Estimates for the Binary AND/OR Process Model. *Technical Report* (Sept., 1985), Dept. of Information and Computer Science, University of California, Irvine.
- [LG85] LEE, R.K.S., GOEBEL, R. Concurrent Prolog in a Multi-process Environment. *1985 Symp. on Logic Programming*, 100-109.
- [LM79] LEVESQUE, H., MYLOPOULOS, J. A Procedural Semantics for Semantic Networks. In *Associative Networks*, N.V. Findler, ed., Academic Press, 1979, pp. 93-120.
- [LM86] LI, P.P., MARTIN, A.J. The Sync Model: A Parallel Execution Method for Logic Programming. *Third Symposium on Logic Programming*, Utah (1986), 223-234.
- [LIND84] LINDSTROM, G. OR-parallelism on Applicative Architectures. *Second Int'l. Logic Programming Conf.*, Uppsala, Sweden (1984), 159-170.
- [LP84] LINDSTROM, G., PANANGADEN, P. Stream-based Execution of Logic Programs. *1984 Symp. on Logic Programming*, 168-176.
- [MARL84] MARLIN, T.J. Emulation of a Data Flow Computer Architecture. *Masters Thesis* (1984), Dept. of Information and Computer Science, University of California, Irvine.
- [McC60] MCCARTHY, J. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Pt. 1. *CACM* V3, n4 (April, 1960), 184-195.
- [McG82] MCGRAW, J.R. The VAL Language: Description and Analysis. *Trans. on Prog. Lang. and Sys.* V4, n1 (Jan., 1982), 44-82.

- [HHKF86] HANDA, K., HIGUCHI, T., KOKUBU, A., FURUYA, T. Flexible Semantic Networks for Knowledge Representation. *J. Information Processing V10*, n1 (1986), 13-19.
- [HILL84] HILLIS, W.D. The Connection Machine: A Computer Architecture Based on Cellular Automata. *Physica V10D* (1984), 213-228.
- [HOARE71] HOARE, C.A.R. Towards a Theory of Parallel Programming. In *Programming Methodology*, D. Gries, ed., 1978, pp. 202-214.
- [HOARE74] HOARE, C.A.R. Monitors: an Operating System Structuring Concept. *CACM V17* (Oct., 1974), 549-557.
- [HOARE78] HOARE, C.A.R. Communicating Sequential Processes. *CACM V21*, n8 (August, 1978), 666-677.
- [HPF86] HONG, Y.-C., PAYNE, T.H., FERGUSON, L.B.O. Graph Allocation in Static Dataflow Systems. *Computer Architecture News, (13th Int'l. Symp. on Comp. Arch.) V14*, n2 (June, 1986), 55-64, Tokyo.
- [HW84] HALIM, Z., WATSON, I. An OR-Parallel Data-driven Model for Logic Programs. *1984 International Workshop on High-Level Architectures*, 1.26-1.36.
- [IM84] ITO, N., MASUDA, K. Parallel Inference Machine Based on the Data Flow Model. *1984 International Workshop on High-level Computer Architecture*, 4.31-4.40.
- [INCE84] INCE, D.C. A Source Code Control System Based on Semantic Nets. *Software - Practice and Experience V14*, n12 (Dec., 1984), 1159-1168.
- [IW87] IOANNIDIS, Y.E., WONG, E. Query Optimization by Simulated Annealing. *ACM SigMod Record*, (Annual Conf., 27-29 May, 1987, San Francisco) *V16*, n3 (Dec., 1987), 9-22.
- [ISRAEL83] ISRAEL, D.J. The Role of Logic in Knowledge Representation. *Computer V16*, n10 (October, 1983), 37-41.
- [JLCH84] JOHNSON-LAIRD, P.N., CHAFFIN, R., HERRMANN, D.J. Only Connections: A Critique of Semantic Networks. *Psychological Bulletin V96*, n2 (1984), 292-315.
- [KGV83] KIRKPATRICK, S., GELATT, C.D., VECCHI, M.P. Optimization by Simulated Annealing. *Science V220* (May 13, 1983), 671-680.

- [PV81] PREPARATA, F.P., VULLEMIN, J.V. The Cube Connected Cycles: A Versatile Network for Parallel Computers. *CACM V24*, n5 (May, 1981), 300-309.
- [PW78] PATERSON, M.S., WEGMAN, M.N. Linear Unification. *J. of Computer and System Science V16* (1978), 158-167.
- [QUIL68] QUILLIAN, M.R. Semantic Memory. In *Semantic Information Processing*, M. Minsky, ed., MIT Press, 1968, pp. 227-270.
- [RAATZ88] RAATZ, S. *Personal communication*. (June, 1988).
- [ROB65] ROBINSON, J.A. A Machine Oriented Logic Based on the Resolution Principle. *JACM V12* (1965), 23-41.
- [ROB71] ROBINSON, J.A. Computational Logic: The Unification Computation. *Machine Intelligence V6* (1971), 63-72.
- [RVS84] ROMEO, F., VINCENTELLI, A., SECHEN, C. Research on Simulated Annealing at Berkeley. *IEEE Int'l. Conf. Computer Design* (1984), 652-657.
- [SAP83] SAPATY, P.S. On the Efficiency of Structural Realization of Operations Over Semantic Networks. *Engineering Cybernetics V21*, n5 (Sept./Oct., 1983), 93-99.
- [SAP86] SAPATY, P.S. A Wave Language for Parallel Processing of Semantic Networks. *Computers and Artificial Intelligence V5*, n4 (1986), 289-314.
- [SE87] SADAYAPPAN, P., ERCAL, F. Nearest-Neighbor Mapping of Finite Element Graphs onto Processor Meshes. *IEEE Trans. on Computers VC-36*, n12 (Dec., 1987), 1408-1424.
- [SGC79] SCHUBERT, L.K., GOEBEL, R.G., CERCONE, N.J. The Structure and Organization of a Semantic Net for Comprehension and Inference. In *Associative Networks*, N.V. Findler, ed., Academic Press, 1979, pp. 121-175.
- [SHAP79] SHAPIRO, S.C. The SNePS Semantic Network Processing System. In *Associative Networks*, N.V. Findler, ed., Academic Press, 1979, pp. 179-203.
- [SHAP83] SHAPIRO, E.Y. A Subset of Concurrent Prolog and Its Interpreter. *Weizmann Institute of Science, Tech. Rep. CS83-06* (1983).

- [McSM79] McSKIMIN, J.R., MINKER, J. A Predicate Calculus Based Semantic Network for Deductive Searching. In *Associative Networks*, N.V. Findler, ed., Academic Press, 1979, pp. 205-238.
- [MDAC87] MCDONNELL DOUGLAS ASTRONAUTICS Co. KNOMES Software Product Specification, DI-MCCR-80025-RMS-KNOMES-V01. Prepared under NASA Contract NAS9-17367.
- [MELL82] MELLISH, C.S. An Alternative to Structure Sharing in the Implementation of a Prolog Interpreter. In *Logic Programming*, Clark & Tärnlund, eds., 1982, pp. 99-106.
- [MM82] MARTELLI, A., MONTANARI, U. An Efficient Unification Algorithm. *ACM Trans. on Prog. Langs. and Syst. V4*, n2 (1982), 258-282.
- [MOLI85] MOLITOR, P. Layer Assignment by Simulated Annealing. *Microprocessing and Microprogramming V16* (1985), 345-350.
- [MT85] MOLDOVAN, D.I., TUNG, Y.-W. SNAP: A VLSI Architecture for Artificial Intelligence Processing. *J. of Parallel and Distributed Computing V2*, n2 (May, 1985), 109-131.
- [NAKA84] NAKAMURA, K. Associative Concurrent Evaluation of Logic Programs. *J. Logic Programming V1*, n4 (1984), 285-295.
- [NAKAG84] NAKAGAWA, H. AND Parallel PROLOG with Divided Assertion Set. *1984 Symp. on Logic Programming*, 22-28.
- [PAT81] PATEL, J.H. Performance of Processor-Memory Interconnections for Multiprocessors. *IEEE Trans. on Computers VC-30*, n10 (October, 1981), 771-780.
- [PB87] POLYCHRONOPOULOS, C.D., BANERJEE, U. Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds. *IEEE Trans. on Computers VC-36*, n4 (April, 1987), 410-420.
- [PKL80] PADUA, D.A., KUCK, D.J., LAWRIE, D.L. High Speed Multiprocessor and Compilation Techniques. *IEEE Trans. on Computers VC-29*, n9 (Sept., 1980), 763-776.
- [PMCA86] PEREIRA, L.M., MONTEIRO, L., CUNHA, J., APARÍCIO, J.N. Delta Prolog: A Distributed Backtracking Extension with Events. *Third International Conference on Logic Programming* (July, 1986), 69-83.

- [VK85] VAGIN, V.N., KIKNADZE, V.G. Deductive Inference on Semantic Networks in Decision-Making Systems. *Soviet J. of Comp. and Sys. Sci.* V23, n2 (March/April, 1985), 1-13.
- [WADK84] WARREN, D.S., AHAMAD, M., DEBRAY, S.K., KALÉ, L.V. Executing Distributed Prolog Programs on a Broadcast Network. *1984 Symp. on Logic Programming*, 12-21.
- [WARR81] WARREN, D.H.D. Efficient Processing of Interactive Relational Database Queries Expressed in Logic. *Research Paper no. 156* (September, 1981), Dept. of Artificial Intelligence, University of Edinburgh.
- [WARR83] WARREN, D.H.D. An Abstract Prolog Instruction Set. *Tech. Report 309*, Artificial Intelligence Center, SRI International (1983).
- [WARR84] WARREN, D.S. Efficient Prolog Memory Management for Flexible Control Strategies. *New Generation Computing V2* (1984), 361-369.
- [WG82] WATSON, I., GURD, J.R. A Practical Dataflow Computer. *Computer V15*, n2 (Feb., 1982), 51-57.
- [WISE82] WISE, M.J. A Parallel Prolog: the Construction of a Data Driven Model. *1982 ACM Symp. on LISP and Functional Programming*, 56-66.
- [WOB186] WONG, W., BIC, L. A Coloring Scheme to Prevent Infinite Recursion in First-order Databases. *Technical Report* (1986), Dept. of Information and Computer Science, University of California, Irvine.
- [WOODS75] WOODS, W.A. What's in a Link: Foundations for Semantic Networks. In *Representation and Understanding*, Bobrow & Collins, eds., Academic Press, 1975, pp. 35-82.
- [WATS88] WATSON, WOODS, WATSON, BANACH, GREENBERG, SARGEANT Flagship: A Parallel Architecture for Declarative Programming. *15th Annual Int'l. Conf. on Computer Architecture* (1988), 124-130.
- [YTK85] YAMAGUCHI, T., TEZUKA, Y., KAKUSHO, O. Parallel Processing of Resolution. *IJCAI* (1985), 1178-1180.

- [SHAS87] SHASTRI, L. A Connectionist Encoding of Semantic Networks. In *Distribute Artificial Intelligence*, M.N. Huhns, ed., Pitman, 1987.
- [TBH82] TRELEAVEN, P.C., BROWNBIDGE, D.R., HOPKINS, R.P. Data-Driven and Demand-Driven Computer Architecture. *Computing Surveys V14*, n1 (March, 1982), 93-143.
- [TC76] Special Issue on Resolution and Automatic Theorem Proving. *IEEE Transactions on Computers VC-25*, n8 (August, 1976).
- [TF86] TAKEUCHI, A., FURKAWA, K. Parallel Logic Programming Languages. *Third International Conference on Logic Programming* (July, 1986), 242-254.
- [TICK84] TICK, E. Towards A Multiple Pipelined Prolog Processor. *1984 International Workshop on High-Level Computer Architectures*, 4.7-4.17.
- [TK84] TAMURA, N., KANEDA, Y. Implementing Parallel Prolog on a Multi-Processor Machine. *1984 Symposium on Logic Programming*, 42-48.
- [TLMS84] TAYLOR, S., LOWRY, A., MAGUIRE, G.Q., STOLFO, S.J. Logic Programming using Parallel Associative Operations. *1984 Symp. on Logic Programming*, 58-68.
- [TR88] TUCKER, L.W., ROBERTSON, G.G. Architecture and Applications of the Connection Machine. *Computer V21*, n8 (August, 1988), 26-38.
- [TURN79] TURNER, D.A. A New Implementation Technique for Applicative Languages. *Software - Practice and Experience V9* (1979), 31-49.
- [TW84] TICK, E., WARREN, D.H.D. Towards a Pipelined Prolog Processor. *1984 Symp. on Logic Programming*, 29-40.
- [UEDA85] UEDA, K. Guarded Horn Clauses. *ICOT Tech. Rep. TR-103* (June, 1985).
- [UT83] UMEYAMA, S., TAMURA, K. A Parallel Execution Model of Logic Programs. *10th Int'l Symp. on Computer Architecture* (1983), 349-355.
- [VEG84] VEGDAHL, S.R. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Trans. on Computers VC-33*, n12 (December, 1984), 1050-1071.